FORTRAN D PARALLELIZING COMPILER: SCHEDULING PHASE

By

GAUTAM S

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

FEBRUARY, 1993

FORTRAN D PARALLELIZING COMPILER: SCHEDULING PHASE

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by
GAUTAM S

to the
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

February, 1993

CERTIFICATE

This is to certify that the work contained in the thesis titled **FORTRAN D PARALLELIZING COMPILER: SCHEDULING PHASE** by **GAUTAM S**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Sanjeev Kumar Aggarwal
Assistant Professor,
Department of Computer
Science and Engineering
IIT, Kanpur

Acknowledgements

I would like to express my deep sense of gratitude to **Dr. Sanjeev Kumar Aggarwal**, my thesis supervisor, for his expert guidance and unstinted encouragement which made working on this thesis an interesting endeavour. The thought provoking discussions with him helped me a lot in channeling my work in the right direction. He introduced me into this exciting and challenging field of parallelizing compilers. It was his wonderful association that made it possible for me to complete this thesis successfully.

My heartfelt thanks go to Ananda R and Rajeev Singh, my project mates, without whose help this thesis would not have seen the light of the day. It was a wonderful experience working with them.

I am extremely greatful to all the members of Kannada Sangha and my class mates who made my stay at IIT Kanpur a memorable one.

I do remember my family for the constant moral support that made me stand through all those crucial moments during the course of this thesis. Their loving memories made me feel I was not far away from home.

Gautam S
IIT Kanpur
Feb 09, 1993.

To my parents

Abstract

There has been a remarkable drift towards parallel processing in the recent past mainly because of the ever increasing demand for speedier computation. The main problem with parallel architectures is our inability to effectively utilize the computing power of such systems. The existing parallel languages fail to present a uniform program model for all architectures. The need to know the machine details to write efficient programs and large investments in traditional software discourage a user from parallel programming. What is needed is an approach which automatically parallelizes the existing programs and executes them efficiently on the given architecture. FRAMES, a restructuring compiler for Fortran 77/Fortran D, extracts the inherent parallelism in the sequential code and maps this parallelism to a shared memory architecture. This thesis deals with the design and implementation issues of scheduling a parallelized program on a shared memory multiprocessor.

Contents

Chapter 1

Introduction

As device technology approaches its limits, it becomes harder to improve the system performance through faster clock rates. Through the replication of processing elements in a coordinated system, one can skip over this performance barrier. Parallel processor systems thus offer a promising and powerful alternative for high performance computing. The amount of parallelism present in scientific software also justifies the use of multiple processors. The need to exploit this potential parallelism and solve problems whose timing requirements cannot be met by even the fastest sequential processor of today forms the main driving force behind the development of parallel systems.

In principle, parallel processing offers a performance which depends only on the application at hand. However, this degree of parallelism cannot be always realized. This is because of our inability to effectively utilize the raw computing power of such massively parallel systems. Overheads due to interprocessor communication/synchronization, processor idle time due to contention of shared resources and uneven distribution of computational load can lead to poor overall performance. Hence it becomes crucial to devise means of efficient utilization of these parallel architectures.

Huge investments in traditional (sequential) software and inadequate support from the parallel languages have made the computing community opt for *restructuring* compilers. Restructuring (parallelizing) compilers extract the hidden parallelism from the sequential code and map the parallelized program on to the given architecture. The concept of restructuring thus relieves the user of the burden of parallel programming. Automatic restructuring however, brings along a host of problems not encountered so far. Problems such as finding an optimal mapping of parallel programs to parallel architectures and minimizing the overheads incurred due to parallel computing are in general intractable. To make matters worse the field of parallel processing is still in its infancy.

This project is an attempt to build such a restructuring compiler for a class of shared memory multiprocessor architectures.

1.1 The Machine model

Multiprocessor systems are becoming increasingly popular because of their flexibility, scalability and high potential performance. They can efficiently execute a large class of programs. In particular, shared memory systems are easier to program. Hence we choose our machine model to be a shared memory multiprocessor. The machine consists of p processors that are connected to a set of global memory modules through a multistage interconnection network (see figure 1.1). The memory can be interleaved and every module is accessible by all the processors. Interprocessor communication takes place through this shared memory. Each processor has its own private memory (a cache or a register file). Every processor can operate asynchronous of other processors. These autonomous processors can be either scalar or vector. For simplicity sake we assume that all the processors are homogeneous and every processor is a scalar. Such a model can be classified as a MES (Multiple Execution array of Scalar

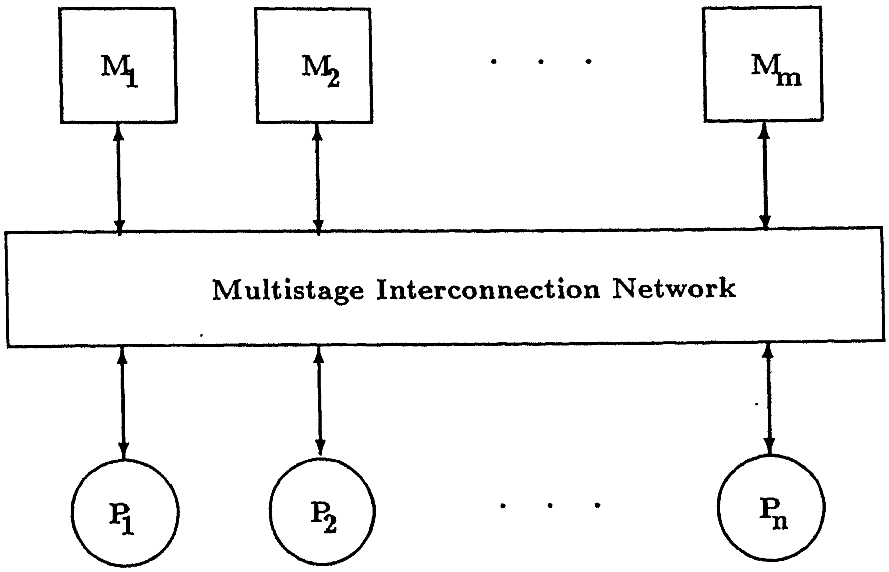

Figure 1.1: Shared Memory Multiprocessor

processors) architecture under Kuck's taxonomy [15].

1.2 The Language assumed

Parallel architectures can become popular only if they can be programmed easily. This needs a language which can express the potential parallelism provided by the underlying architectures. Many researchers feel that the communication overheads associated with parallel processing can be greatly reduced if data is properly placed on the different processors. A parallel language which supports data distributions would serve the purpose. **Fortran D**[6] is one such language specially designed for a class of distributed memory multiprocessors. Fortran D is a version of Fortran 77 enhanced with data distribution specifications. Appendix A gives some details of Fortran D relevant to data distributions. More about Fortran D can be found in [6].

1.3 Structure of the project

Restructuring compiler FRAMES (Fortran D Restructuring Aid for MES) is being developed for a shared memory multiprocessor. The main objective is to accept a program written in Fortran D, parallelize the sequential portion of the program (wherever possible), and schedule this restructured code on the given architecture. We present the user with a semantically equivalent high level program which contains explicit communication and synchronization instructions. The project is inherently modular by design and the different blocks are as shown in the figure 1.2.

Phase 1 parses the input program as done in any other compiler. In addition, global data flow analysis and machine independent optimizations are also performed. **Phase 2** is the restructuring phase. This phase identifies the constructs that can be parallelized and such constructs are restructured. Care is taken to ensure that the semantics of the original program are not violated. FRAMES allows the user to view the changes made to his program at this stage (BACK END 1), and direct the compiler to parallelize/serialize certain constructs to improve the performance. **Phase 3** is architecture dependent. The problem of mapping the parallelized program to the underlying architecture is handled here. The first two stages are thus insulated from the machine details. The parallelized program is broken down into a set of disjoint tasks (either dependent or independent). A task could be informally defined as a program fragment. These tasks are then scheduled on the processors for parallel execution. The work is mainly centered around exploiting parallelism in loops. An optimal assignment of processors to loops is performed. Based on the scheduling decisions made, we generate a semantically equivalent high level program which reflects these scheduling decisions (BACK END 2). Presenting this 'transformed' program to the user enables him to study the effect of scheduling decisions on the program

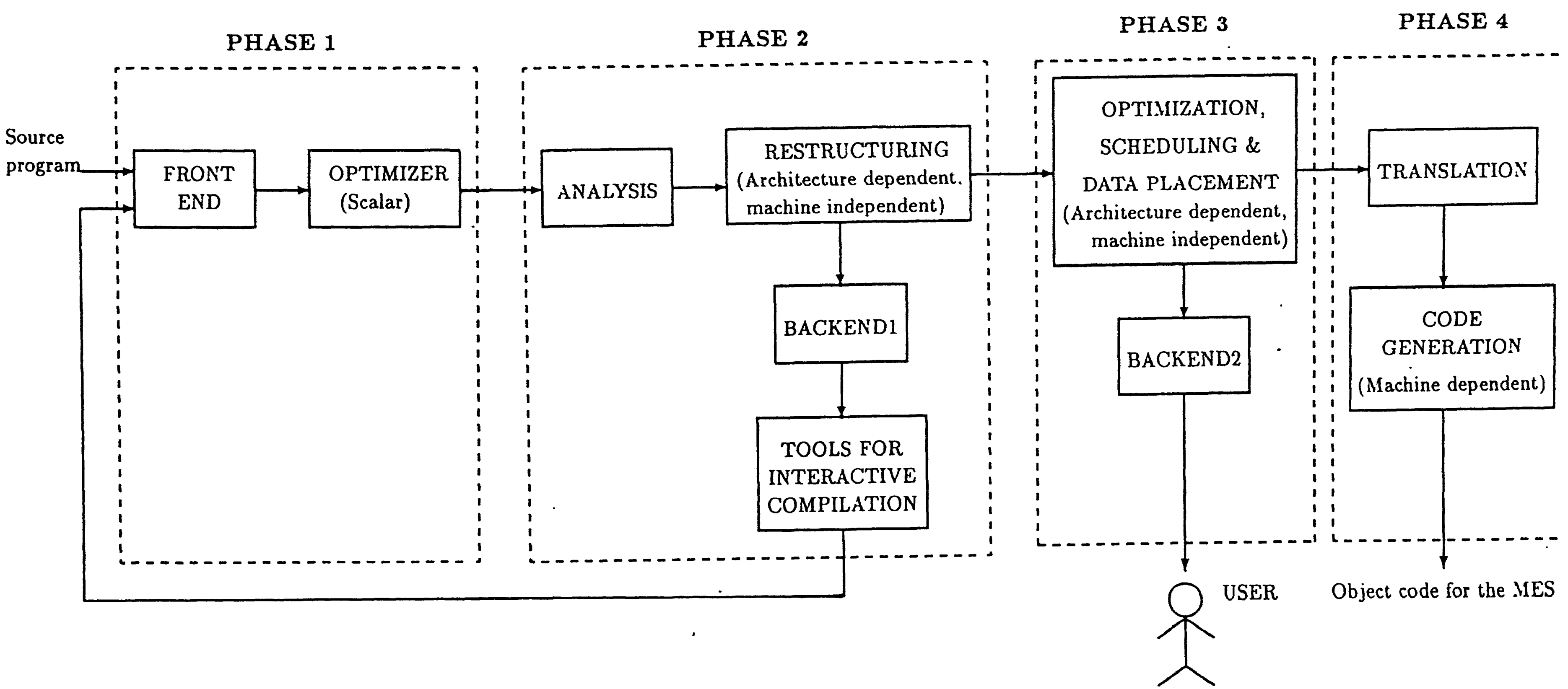

Figure 1.2: Structure of the Project

and give directives to the compiler for improving the performance. **Phase 4** is completely machine dependent. The machine code for the restructured and scheduled program is generated here. FRAMES can be viewed as any one of the following.

- A Fortran D to Fortran D converter (Phase 1 + Phase 2 + feed back). Since Fortran 77 is a proper subset of Fortran D, one can view FRAMES as a Fortran 77 to Fortran D restructurer also.
- A *black box* that maps a Fortran D program to a given architecture (the entire compiler)

This thesis deals with the problem of partitioning and scheduling parallel programs described in Phase 3.

1.4 Outline of the thesis

The problems of packaging and scheduling of parallelism and the issues involved are discussed in the ensuing chapters. Although the techniques discussed are for shared memory models, they can be applied to other models as well. The techniques described are independent of the language features. To understand the concepts, we present the examples in standard Fortran. Constructs not found in Fortran 77 are explained as and when they are encountered.

Chapter 2 addresses the issues involved in partitioning a program and scheduling the partitioned program. A survey of the existing techniques is made.

Since parallel loops account for a major portion of parallelism in numerical programs, it is worth examining the problem of scheduling such constructs in greater detail. **Chapter 3** focuses on the problem of efficient scheduling of parallel loops and the issues involved. Some popular scheduling strategies for

efficient execution of parallel loops are also looked at.

When multiple processors work on a single program, they have to execute in a coordinated fashion. Synchronization between the processors is required to satisfy the data dependences. The problem of generating a target program which includes all these details is dealt with in **Chapter 4**. Some techniques which make use of the architectural details in improving the program performance are also investigated.

Implementation details are covered in **Chapter 5**.

Chapter 6 presents some test results and suggestions for further improvements are made.

A brief summary of Fortran D can be found in **Appendix A**.

Chapter 2

Packaging and Scheduling of Parallelism

Given a parallel/vector program, written independently or transformed into parallel/vector form by a parallelizing/vectorizing compiler from its sequential counterpart, the main issue is that of packaging and scheduling of the parallelism present in that program. If a program is to be executed on a parallel machine, then each processor should know which part(s) of the program it is going to execute. This requires partitioning of the program into disjoint modules and assigning these modules to the individual processors. The modules should be defined and scheduled in such a way that the overall execution of the program is as fast as possible. We cannot exploit parallelism more than what the machine can offer. The underlying architecture thus puts a limit on the extent to which program parallelism can be exploited. This chapter addresses the issues of parallelism packaging (program partitioning) and scheduling.

```
DO I = 1,100
   X(I) = A(I)+B(I)
ENDDO
DO J = 1,100
   Y(J) = A(J)-B(J)
ENDDO
DO K = 1,100
   SUM = SUM + X(K)/Y(K)
ENDDO
```

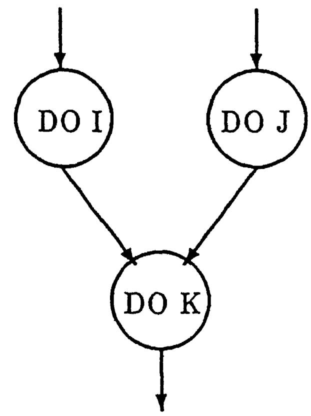

Figure 2.1: An example program fragment and the corresponding task graph

2.1 Program Partitioning

Program partitioning can be informally defined as breaking a program down into a set of disjoint tasks and define the data and control dependences [1] between the tasks. A task can be viewed as a program module that executes on one or more processors. The dependence between the tasks is usually represented in the form of a directed graph, the *task dependence graph* or more commonly the task graph. The nodes in the task graph correspond to the tasks. An edge between two tasks implies the existence of one or more dependences between them. During execution, a task cannot start unless all preceding tasks on which it depends have completed execution. We assume that every task graph has a single entry node (with no predecessors) and a single exit node (with no successors) such that every node (other than the entry and exit nodes) lies on at least one path from the entry node to the exit node. An example program fragment and the corresponding task graph are as shown in the figure 2.1.

Tasks are either serial or parallel depending on whether they can execute on

[1] A *data dependence* exists between two entities if they access the same memory location, at least one being a write. A *control* dependence exists between a conditional statement and all statements that are under the control of one of its branches

one or more processors. i.e. a serial task would request for only one processor during execution while a parallel task would request for more processors. Irrespective of whether a task is serial or parallel, the processor executing it will execute sequentially. Parallel execution of a task now means decomposing it into a set of serial entities and then executing them on different processors simultaneously. These serial entities are referred to as *processes.* Processes can be defined by the compiler either before execution (static) or dynamically during program execution. The number of available processors would decide the number of processes into which a task should be decomposed. Each task would therefore get as many processors as the processes defined in it. The issues involved in partitioning are thus closely related to those involved in scheduling.

2.1.1 Static v/s Dynamic partitioning

Program partitioning can be done either at compile time (**Static**) or at run time (**Dynamic**). In a fully dynamic environment (eg. dataflow), partitioning is implicit and depending on the execution model, a task can range from a single instruction to a set of instructions. Static partitioning is an attractive approach since it exploits the available information about the program parallelism and considers the overheads involved during execution and other performance factors.

2.1.2 Task granularity v/s Load balancing

An important issue in partitioning is of task granularity. By granularity we mean the size of the task. Larger the task granularity lesser is the number of tasks to be dispatched for execution and hence the penalty incurred during dispatching. But this may result in an uneven load distribution on the processors. Uneven load on the processors can greatly affect the program performance. If tasks

are of smaller granularity, load balancing becomes less of an important factor. However, if the overheads associated with dispatching a task become significant, it would be wise to have tasks of larger granularity. These overheads impose a minimum size on tasks, below which the *speedup* [2] becomes rather a 'slowdown'. This minimum task size is called the *critical task size.*

2.1.3 Critical task size

The issue of interest here is the estimation of the critical task size. When a parallel task is distributed on to several processors at run time, it incurs a penalty or overhead that limits the degree of exploitable task granularity. Run time overhead includes several activities that do not occur during serial execution. For example, processors involved in executing a parallel task may have to wait for each other before proceeding to other jobs. In some sense the processors have to cross a 'barrier' before executing other tasks. Also processors need to enter critical sections while accessing global data [10]. These overheads make it inefficient to execute in parallel, small tasks. If the task is not large enough to amortize the overhead, we may end up with a parallel execution time which is much larger than the serial execution time. If a parallel task of size T which when executed on a system with p processors, suffers an overhead O_s, then speedup is given by

$$S_p = \frac{t_{serial}}{t_{parallel}} = \frac{T}{\frac{T}{p} + O_s}$$

assuming that the task uses all the p processors.

In order to have a speedup of at least 1, we must have

$$T \geq \frac{T}{p} + O_s \Rightarrow T \geq \frac{p * O_s}{p - 1}$$

[2]Speedup is a measure of the gain in execution time due to the use of multiple processors. Speedup is defined to be the ratio of parallel execution time to serial execution time.

which gives the critical task size as a function of overhead and the number of processors.

A compromise would be to allow the compiler to statically define the tasks of large granularity and these tasks could be later decomposed at run time, if it helps in improving load balancing.

2.2 Static Program Partitioning

The problem of static program partitioning and the issues involved are discussed in this section.

2.2.1 Methods for program partitioning

There are two methods for partitioning a program. In the *top-down* approach, we start off assuming that the entire program is a single task. Then the program is decomposed into smaller tasks recursively, following some rules. An alternative to this is the *bottom-up* approach. Here, each statement is assumed to be a task. Then the tasks are merged together to form larger tasks. Splitting/merging is done so that the resulting tasks have as few inter-task dependencies as possible and there are enough tasks to keep the processors busy during execution.

In bottom-up approach, we start with the lowermost representation of the program, possibly the data dependence graph, in which case information about the global structure of the program is readily available. Hence, local optimizations can be easily performed during merging. This is not the case with top-down approach because information about the internal structure of the program is not available and it becomes expensive to perform local optimizations.

Partitioning is usually performed in such a way that the resulting task graph is a directed acyclic graph (DAG). This means that a loop cannot span over

more than one task. While working with imperative languages like C, Pascal and Fortran we can follow a simple heuristic approach which partitions the program depending on the syntax of the language; This approach uses the 'natural' boundaries in the program to define tasks. For example, a loop nest is defined to be a task. A piece of straight line code is also considered to be a task. If we assume that only structured programming is allowed, then the task graph generated by this heuristic will in fact be a DAG.

2.2.2 Communication v/s Parallelism

Communication and parallelism forms another issue while defining tasks. Though communication overheads occur at run time, we need to consider this at compile time itself. Communication would take place between two tasks U and V, if there exists a dependence between them. During execution, if U and V run on different processors either concurrently or at different times, data computed by one must be sent to the other. On the other hand we can execute both the tasks on the same processor, one after another. This would not involve any explicit communication overhead but clearly we are sacrificing on parallelism. By preventing two tasks from executing on different processors, we are in some sense merging the two. Merging is hence done only when the saving in interprocessor communication outweighs the loss of potential parallelism. Clearly, there is a tradeoff between the two.

Communication overheads are less critical in case of shared memory systems. Hence simple heuristics can be as effective as any optimal or near optimal partitioning scheme. However, this is not the case with distributed memory machines. Here interprocessor communication has to take place explicitly through send/receive messages and these messages may have to be routed through intermediate processors before they reach their destinations. Because of these factors

communication overheads become quite significant and more sophisticated algorithms have to be used to perform partitioning.

In most of the cases parallelism and communication form conflicting objectives to be achieved (i.e. optimizing one, counteroptimizes the other). The problem of simultaneously maximizing parallelism and minimizing communication overhead can be mapped to a network flow problem. This problem has been studied extensively and found to be NP-complete [16].

2.3 Task Scheduling

After breaking a parallel program into a set of interdependent (possibly independent) tasks, the question that arises is how to schedule the different processors to execute these tasks so that the overall execution is as fast as possible. Even in its simplified formulations, the problem of scheduling is found to be NP-complete. Optimal polynomial time solutions have been found for some special cases but these cases do not occur frequently in practice. Also problems such as scheduling, minimizing interprocessor communication and synchronization are often found to be architecture dependent. Finding universal solutions to such problems would be almost near to impossible. Because of the disparity in the existing machine organisations, a solution efficient for one can be very inefficient for another. Still it is possible to find general solutions for large classes of machine architectures which share some common features.

2.3.1 Static v/s Dynamic Scheduling

Program scheduling can be broadly classified as static and dynamic. In static schemes, processors are assigned to the different tasks deterministically before execution. That is, during execution every processor knows exactly which task to execute next. The main advantage with static schemes is that they do not incur

any run time overheads. Also they need less operating system support. Since global information is available, we can deliberately keep some processors idle so that they can be better utilized at a later stage. All said and done, static schemes can generate optimal schedules only if execution times of the tasks are known accurately. Even in simple cases, it becomes difficult to estimate execution times accurately because of memory interferences, delays in the network and operating system delays.

Dynamic schemes schedule the processors to execute the tasks nondeterministically at run time. The advantage with these schemes is their flexibility. Even with less precise information about the program, they can perform reasonably well. Since scheduling decisions have to be made dynamically at run time, they tend to slow down the execution. So it becomes essential for these schemes to make simple and fast decisions. A direct implication is that these schemes cannot afford to look at the global structure of the program. Typically, dynamic scheduling algorithms make decisions based on local information. Since these algorithms ignore global program information and general topology of the task graph, the generated schedules can be very inefficient. However, in real cases the resulting schedules come very close to the optimal.

Dynamic scheduling can either be performed through centralized control or distributed control. In the former, a global unit is responsible for making scheduling decisions. In the latter, processors themselves decide which tasks to execute next and fetch them from a common pool of ready to execute tasks. It is not yet clear, which of the two can perform better than the other.

2.3.2 Dedicated v/s Multiprogramming Mode

A more basic issue is that of the operating mode of the parallel processor systems; whether the system has to operate in dedicated mode or in the multipro-

gramming mode. This greatly influences the design of scheduling algorithms. For real time applications, dedicated mode of operation is most suitable; otherwise multiprogramming mode is appropriate. Although the main goal of parallel processing is to minimize the execution time of application programs, huge investments in these parallel architectures make efficient utilization of resources also an important issue. Achieving both the goals is not always possible. If parallel processing becomes cheaper, then efficient utilization of resources will become less of a deciding factor. In the following discussion we assume purely dedicated environments where minimizing execution time becomes the main goal to be achieved.

2.3.3 Assumptions

An assumption being made is that a task once scheduled on a processor will not be preempted. It is practical to assume that tasks(more precisely, processes) are not preempted because this would involve significant overheads at run time. The operating system has to do a lot of bookkeeping work during preemption and this could possibly slow down the execution of the program.

It is also possible to exploit parallelism at the statement level. Exploiting fine grain parallelism is however highly machine dependent and is not considered here.

Many scheduling algorithms produce schedules from a *layered* task graph. A layered task graph is a task graph where all nodes that are independent of each other are marked with same label to form a *layer*. The maximum distance of a node from the entry node is defined to be its label. Since the task graph is a DAG, the nodes can be easily labeled by performing a simple graph search. An example task graph and the corresponding layered graph are shown in the figure 2.2.

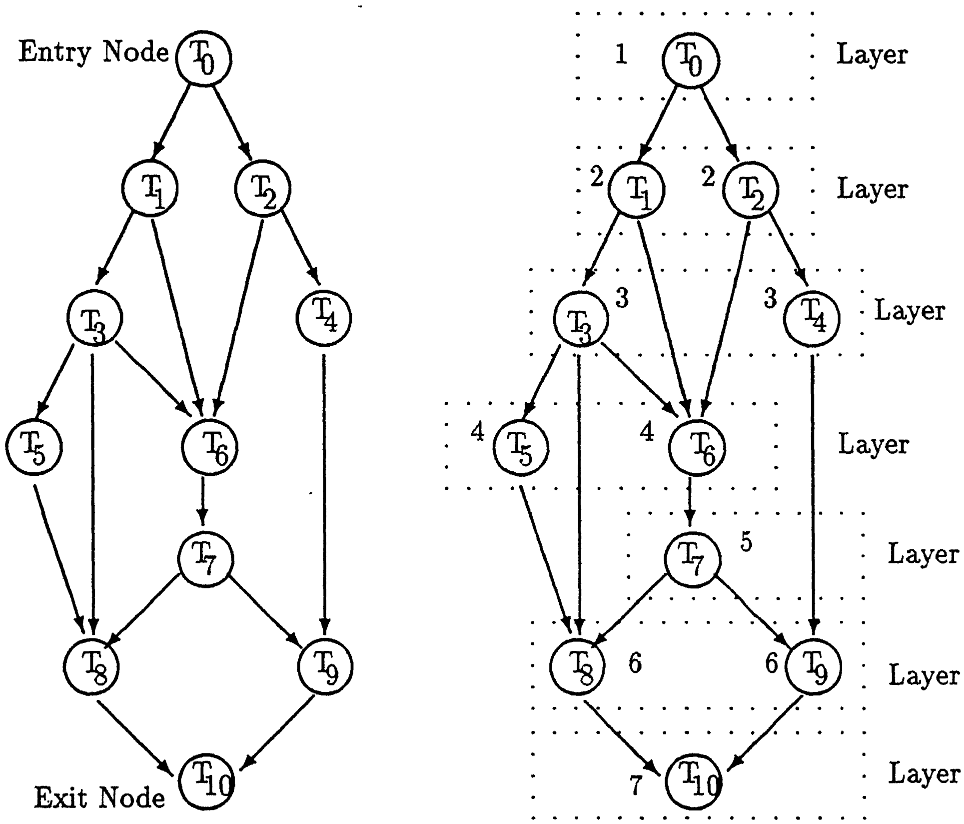

Figure 2.2: A Task graph and the corresponding layered graph

2.3.4 Scheduling of Independent Tasks

In this section we consider the problem of assigning a certain number of processors to a set of independent tasks. Independent program modules are frequently found in scientific code. In some numerical programs, as many as 13 independent parallel loops have been found. Since no dependences exist between any pair of tasks, all tasks can be executed simultaneously.

Suppose we are given a set $S = \{M_1, M_2, \cdots M_m\}$ of independent tasks and

a system with $p \geq m$ processors; each program module M_i can request for a maximum of p_i processors ($i = 1, 2, \cdots m$). The problem is to find an optimal processor assignment to these tasks. This problem can be solved in polynomial time using the algorithm **OPTAL** (explained in the next chapter).

We define an allocation function $G_i(q)$ as follows.

$G_i(q) = T_i^q$, for $q = 1, 2, \cdots p$

where T_i^q is the parallel execution time of M_i with q processors.

for $1 \leq i < m$

for $q = 1, 2, \cdots p$

$G_i(q) = \min\{\max(T_{i+1}^r, G_i(q - r))/r = 1, 2, \cdots q\}$

The idea is to first assign all the p processors to the task M_m. We then *steal* some processors from this task and assign them to the adjacent task so that the overall execution time of the two is the minimum. We continue this process till all the tasks are processed. Note that while computing the minimum execution time of S with p processors we automatically construct the processor allocation vector.

Computing the parallel execution time of M_i with q processors accurately for all possible values of q would be very expensive. Instead, we can find a close approximation to T_i^q inexpensively by computing the serial execution time T_i^1 and then compute T_i^q as

$$T_i^q = \frac{T_i^1}{q} \text{ , for } q = 2, 3, \cdots p$$

This is very accurate if the task is a parallel loop, which often is the case.

If there are more independent tasks than processors, then the problem of computing an optimal processor assignment becomes NP-complete. In such cases heuristics that can give near optimal solutions in polynomial time are adopted.

2.3.5 Scheduling of Complete Task Graphs

Because of the complexity of the scheduling problem, heuristics are often sought after. A family of heuristics can be found in the literature and the more popular ones are the list scheduling algorithms. The basic idea is to arrange the nodes of a given program task graph in a priority list and then assign the tasks with highest priority to processors as and when they become idle.

Some common criteria in assigning priorities are as listed.

1. Give priority to tasks that belong to the *critical path* [3].

2. Give priority to tasks that have longest execution times.

3. Give priority to tasks that have largest number of immediate successors.

4. Give priority to tasks that have successors with long execution times.

Some heuristics compute a composite priority by taking the weighted average of all the above listed priorities. Usually the tasks that lie on the critical path are given absolute priority. The CP/MISF method (Critical Path / Most Immediate Successors First) is claimed to be the best heuristic so far [11]. These heuristics can be 'tuned' to obtain required performance by adjusting the weights associated with the individual priorities.

Scheduling is then performed layer by layer. i.e. tasks in the $(i+1)$-th layer cannot start until all tasks of layer i have completed. Note that tasks within a layer are independent of each other and hence can be executed simultaneously. Figure 2.3 gives a detailed algorithm.

[3] *Critical path* is a path from the entry node to the exit node having the longest execution time.

Input: A p-processor machine and a layered task graph G with each task t_i requesting for $r_i < p$ processors.

Output: A processor assignment to G.

Method:

1. *while* graph G not completely processed

1.1 Arrange the tasks of the next layer in the increasing order of priorities in a priority list L.

1.2 *while* list L not empty

1.2.1 Select first k tasks such that $\sum_{i=1}^{k} r_i \leq p \leq \sum_{i=1}^{k+1} r_i$

1.2.2 Assign r_i processors to each selected task t_i.

1.2.3 Delete these tasks from L.

endwhile

endwhile

2. Output the processor assignment to G

Figure 2.3: A List Scheduling Heuristic

Chapter 3

Scheduling of Loops

Loops form a rich source of parallelism in scientific code. To take the advantage of the capability of parallel machines, this parallelism must be effectively utilized. Because of the prevalence of such parallel loops, optimal parallel loop scheduling has received considerable attention. Efficient scheduling algorithms can dramatically improve the system performance and hence it becomes reasonable to focus our attention on efficient scheduling of parallel loops.

3.1 Notations

In a parallel program, we observe the following three types of loops; `DO` loops, `DOALL` loops and `DOACR` loops.

Any loop (`DO`, `DOALL` or `DOACR`) has the following form.

```
DO I = L,U,S
   {B}
ENDDO
```

where `L` and `U` are the lower and upper bounds of the loop iteration variable (or loop index) `I` and `S` is the loop stride through which `I` is incremented after

every iteration; B is the loop body which can consist of straight line code (a set of assignment statements) or another loop itself. To simplify our notations, each loop is assumed to be normalized. i.e. its iteration space is of the form $[1 \cdots N], N \epsilon\ \mathcal{Z}^{+}$ and S is unity. A normalized loop would now look like

```
DO I = 1,N,1
   {B}
ENDDO
```

The DO loop is completely serial. i.e. a definite order is imposed on the execution of the different iterations of the loop. At the other end of the spectrum, we have the DOALL loop. DOALL loop is completely parallel in nature; the different iterations of the loop can be executed in any order. In other words, all iterations of the DOALL loop can be executed concurrently. A DOACR loop can be informally defined as a parallel loop in which data dependences allow for partial overlap of the execution of successive iterations. i.e. if iteration i starts at time t on a processor, iteration $i+1$ cannot start until a time slice d has elapsed after iteration i has started execution, where d is a constant, called the *delay*. If b is the serial execution time of the loop body, then the ratio d/b is defined to be the percentage of overlap. When $d = b$, the loop is completely serial, while if $d = 0$, the loop is completely parallel. DO and DOALL are therefore special cases of DOACR loops.

If there is no other loop contained in the loop body, the loop is called an innermost loop. Other loops are called outer loops. In a nested loop, an individual loop can be enclosed by many outer loops. The nest level of an individual loop is equal to one plus the total number of enclosing outer loops. Nest depth of a loop nest is the maximum nest level of the loops in the nest.

A one-way nested loop (perfectly nested) of nest depth k has exactly one loop at each nest level i, $i = 1, 2 \cdots k$ (see figure 3.1). A perfectly nested loop of

```
DO I1 = 1,N1
   DO I2 = 1,N2
      ...
        ...
        DO Ik = 1,Nk
           {B}
        ENDDO
        ...
      ...
   ENDDO
ENDDO
```

Figure 3.1: A perfectly nested loop of depth k

depth k is denoted by $L(N_1, N_2, \cdots, N_k)$ where N_i is the number of iterations of the loop at nest level i.

A loop is m-way (multi-way) nested if there exist at most m disjoint loops at some nest level.

Nested loops that contain combinations of DO, DOALL and DOACR loops are called *hybrid*.

Because of the well defined structure of loop nests, scheduling of such constructs becomes less complicated. Since parallel loops account for the greatest percentage of parallelism in numerical programs, designing low overhead methods for the efficient allocation of processors to loops becomes crucial to machine performance.

3.2 Scheduling goals

Any scheduling scheme would aim at achieving the following two objectives.

1. Keep all the processors as busy as possible.

2. Run time overhead must be kept minimal.

However, we cannot optimize one without considering the other. The fundamental tradeoff in any scheduling policy is that of balancing processor workloads v/s minimizing scheduling overhead. Load balancing refers to the finish time of all the processors executing a parallel program. Suppose all processors in a system start executing at the same time. The latest to finish would clearly determine the parallel execution time. An ideal situation would be one where all the processors finish execution at the same time. We say the work load is evenly distributed among all the processors and hence load balancing is optimal. This issue is not independent of task granularity and partitioning.

Consider, for example, a parallel program being scheduled using a sophisticated approach which achieves perfect load balancing but totally ignores the scheduling overheads and then a naive scheduling scheme which focuses on minimizing overheads disregarding load balancing. The sophisticated approach may give balanced load with all the processors finishing at time t_b. The latter may have a highly unbalanced load with the last processor finishing at time t_u, such that $t_u < t_b$. This may happen due to the significant overhead incurred by the sophisticated approach. The naive scheme has achieved an overall better execution time and there is still scope for load balancing. A better approach would therefore be to have an algorithm that incurs less overhead and achieves better load balance.

Unbalanced load occurs several times during the execution of a single program. This would result in some processors idly waiting for others to finish. This is particularly true in the presence of barrier synchronization. Typically, a barrier will be set at the end of every parallel loop in a program and all processors must clear this barrier before proceeding to other jobs. Another issue which needs attention is whether scheduling has to be performed at compile time (**Static**

scheduling) or at run time (**Dynamic scheduling**).

Static allocation schemes involve minimum run time overheads since scheduling decisions would have been made by the compiler before execution begins. During program execution, each processor knows exactly which task to execute. A scheduling scheme is called dynamic when processor allocation is performed during execution. Here the time taken by decision making process is reflected in the execution time. Dynamic approaches involve considerable overhead which makes parallel execution of small loops impractical. Hence any run time scheduling scheme should aim at making simple and fast decisions. However, static scheduling techniques cannot generate optimal schedules if loop bodies have variable execution times (i.e. loop bodies contain branch statements) or if the loop bounds are not known at compile time, which unfortunately is the case most of the time. Dynamic scheduling seems to be a better alternative in such situations.

The rest of the chapter makes a brief survey of the existing scheduling techniques. Throughout this chapter the terms scheduling, processor allocation and processor assignment are used interchangeably.

3.3 Scheduling techniques

This section reviews some popular dynamic scheduling techniques.

3.3.1 Static Chunking

A straight forward approach in exploiting parallelism in `DOALL` loops is by allocating chunks of iterations (iteration block) to the processors, often referred to as *static chunking* or *chunk scheduling*. Thus in a system with p processors, a `DOALL` loop with $n > p$ iterations would be executed by scheduling a chunk

of $\lceil n/p \rceil$ successive iterations to every processor (except for the last one, possibly). Therefore, processor i would execute the iterations $i * (\lceil n/p \rceil) + 1, i * (\lceil n/p \rceil) + 2, \cdots (i+1) * \lceil n/p \rceil$. We could also perform static chunking by allocating successive iterations to successive processors. i.e. processor i would now execute the iterations $i, p+i, 2p+i, \cdots (\lceil n/p \rceil - 1) * p + i$. The latter can also be used to execute DOACR loops efficiently, as successive iterations now go to different processors. Since chunks of $\lceil n/p \rceil$ iterations are dispatched at a time, there can be at most p dispatches. If a DOALL loop of 1000 iterations is executed on 4 processors, then each processor would execute 25 successive iterations (see Table I).

This is the best method as far as minimizing scheduling overheads is concerned. Static chunking works well as long as the iterations have constant execution time. If the loop body contains branches, then iterations may have variable execution times. In such cases, static chunking may result in highly unbalanced load on the processors and the overall execution time can be far worse than the optimal. This method is favored only when the scheduling overheads become significant.

3.3.2 Self Scheduling

Alternatively, we can schedule the iterations one at a time, a strategy called *self scheduling.* With self scheduling, a processor obtains new iteration whenever it becomes idle. Thus for a DOALL loop with n iterations, there will be n dispatches as opposed to static chunking. Since all accesses to the loop indices are, by necessity synchronized, every time a processor obtains an iteration, it has to enter the critical section. Roughly speaking, if p processors are involved in executing the loop, then each processor gets $\lceil n/p \rceil$ iterations. Because self scheduling assigns one iteration at a time, it is the best scheme as far as load

balancing is concerned. However, perfectly balanced load is meaningless if the overheads incurred to achieve it exceeds a certain threshold.

The characteristics of the iterations hence determine which scheme performs better. Self scheduling is appropriate if there is a large variation in the execution times of the different iterations and the loop body is large enough to overshadow the overheads involved during scheduling; constant length, fine grained iterations on the other hand favor static chunking.

Static chunking and self scheduling are the two extremes. Between the two lie schemes that attempt to strike a compromise by achieving better load balancing with less overheads. Such schemes schedule iterations in chunks of size greater than one but less than $\lceil n/p \rceil$. These schemes usually have variable chunk sizes although schemes with constant sized chunks are not uncommon.

It is practical to assume that the processors involved in the execution of a loop nest start executing the iterations at different times. This is a valid assumption because the processors will be busy executing other parts of the program, before they start executing the loop nest. It is a usual practice to set a barrier at the end of a parallel loop and if the proceesors executing it finish at different times, then the processors may have to wait for significant amount of time before proceeding to other tasks. Hence it would be desirable if the scheduling schemes take this into consideration and generate schedules such that the processors terminate approximately at the same time.

3.3.3 Guided Self Scheduling

Guided Self Scheduling (**GSS**, in short) [15] is one such scheme which aims at achieving better load balance with less scheduling overheads. This scheme *guides* the processors on the amount of work they choose. It also considers the uneven starting times of the processors. GSS employs variable sized chunks

which decrease in size. The principle behind this is by allocating large chunks in the beginning, frequent dispatching is avoided and hence the overheads associated with them. The small chunks at the end try to balance the load across all the processors and smoothen the uneven finishing times of the processors. Every incoming idle processor follows the principle that it should leave enough iterations for the remaining $p-1$ processors, in case they all become idle at the same time.

The chunk size G_i scheduled on the next idle processor is given by $\lceil R_i/p \rceil$ where R_i is the number of iterations left to be executed.

Thus we have,

$$\begin{aligned} R_0 &= n \\ G_i &= \left\lceil \frac{R_i}{p} \right\rceil ; \; R_{i+1} = R_i - G_i \\ \Rightarrow \; G_i &= \left\lceil \left(1 - \frac{1}{p}\right)^i * \frac{n}{p} \right\rceil \end{aligned}$$

For a DOALL loop with 1000 iterations, the iteration blocks are as shown in the Table I when executed on a 4-processor machine.

It has been proved that for constant length iterations and uneven processor starting times, processors that are scheduled under GSS finish executing within one iteration of each other and the obtained schedule is optimal [15]. GSS goes through $O(p * \log(n/p))$ synchronization points in the worst case. By synchronization points, we mean the number of times processors enter critical region to obtain the next chunk of iterations.

If the iterations have large variance in their execution times, then the schedule obtained from GSS can be far from optimal. We observe that the chunk size decreases exponentially. Because of this the initial chunks scheduled by GSS will be quite large and there may not be enough iterations left to smooth over

the uneven finishing times of even the first p chunks.

Because of its simplicity, GSS can be efficiently implemented to generate schedules at run time. It can even be incorporated in hardware.

3.3.4 Factoring

Factoring [9], another dynamic scheduling scheme, addresses the problem of handling iterations with execution time variance. Here iterations are scheduled in batches of p equal sized chunks. The total number of iterations per batch is a fixed ratio of those remaining and hence the name factoring.

To achieve an overall optimal finishing time, for each batch scheduled there must be enough work left to smooth over the uneven finishing times of that batch. The idea is, if the larger chunk sizes of the earlier batches are estimated conservatively, then the smaller chunk sizes of later batches are not so critical. Assuming that the iteration execution time is a random variable with a bell shaped distribution, we obtain a normal distribution for the chunk execution time (which is a sum of random variables). To have a high probability of even finishing times, no more than half of the remaining iterations should be scheduled in every batch [9].

The chunk size F_i is given by,

$$
\begin{aligned}
R_0 &= n \\
F_i &= \left\lceil \frac{R_i}{2p} \right\rceil ; \; R_{i+1} = R_i - p * F_i \\
\Rightarrow \; F_i &= \left\lceil \left(\frac{1}{2}\right)^{i+1} * \frac{n}{p} \right\rceil
\end{aligned}
$$

where R_i is the number of iterations left to be executed.

If a `DOALL` loop with 1000 iterations is executed on 4 parallel processors, then Factoring would schedule the iterations as shown in Table I.

Factoring is hence a general form of GSS. If each batch scheduled consists of only one chunk, then factoring reduces to GSS. Factoring requires lesser number of synchronization points as sizes of chunks to be scheduled are decided in batches of p. Intuitively factoring should perform better.

Scheme	Sample Chunk Sizes (N = 1000, P = 4)
Static Chunking	250 , 250 , 250 , 250
Self Scheduling	1 , 1 , 1 , 1 , . . .
GSS	250,188,141,106,79,59,45,33,25,19 14,11,8,6,4,3,3,2,1,1,1,1
Factoring	125,125,125,125,62,62,62,62,32,32,32,32 16,16,16,16,8,8,8,8,4,4,4,4,2,2,2,2,1,1,1,1

Table I: Table of sample chunk sizes

3.4 Static Loop scheduling

As stated earlier, static scheduling schemes can be very efficient if the loop iterations have constant execution time and the loop bounds are known. Also they incur minimum scheduling overheads. Earlier work done in this area exploited parallelism in singly nested loops only. In case of nested loops, all the processors were assigned to the innermost loops. This was partly justifiable because of the small number of processors in the machines. Parallelism in one dimension was

sufficient to use up all the processors. However with the advent of supercomputers with large number of processors, this is no longer true. Proper utilization of the proceesors to exploit parallelism as much as possible becomes vital to system performance. Also, in scientific code, it is often the case where several nested loops can be executed at once, assuming unlimited number of processors. With such nested loops, parallelism can still be better utilized with limited number of processors if they are allocated to more than one loop.

Optimal processor assignment : Processor assignment to a loop nest can be informally defined as the problem of assigning the processors to the individual loops in the nest so that the parallel execution time of the construct is minimized. These techniques partition a p-processor machine hierarchically into sets of processors and assigns the different sets to different loops.

Note: The schedules obtained from static allocation schemes in general, may not be necessarily optimum when compared with schedules obtained dynamically. The optimality discussed here is in the context of static scheduling.

An arbitrarily nested loop of nest depth k can be uniquely represented as a k-level tree. The leaves of the tree correspond to blocks of assignment statements (BAS's) in the loop nest and the intermediate nodes correspond to the individual loops. Clearly, intermediate nodes at level m correspond to the loops at nest level m. The total number of nodes in such a tree is $\lambda + \mu$ where λ is the number of individual loops in the nest and μ is the number of BAS's. We number the loops in the nest increasingly in lexicographic order. Figure 3.2 depicts an example loop nest and its tree representation. For convenience sake we consider the loop bodies also to be loops with a single iteration.

If a `DOALL` loop with N iterations is executed on a p-processor machine, then each processor (but possibly the last) is assigned $\lceil N/p \rceil$ iterations. Hence par-

```
DO I1 = 1,N1
   {a}
   DO I2 = 1,N2
      DO I3 = 1,N3
         DO I4 = 1,N4
            {b }
         ENDDO
      ENDDO
      DO I5 = 1,N5
         DO I6 = 1,N6
            {c}
         ENDDO
         DO I7 = 1,N7
            {d}
         ENDDO
      ENDDO
      {e}
   ENDDO
ENDDO
```

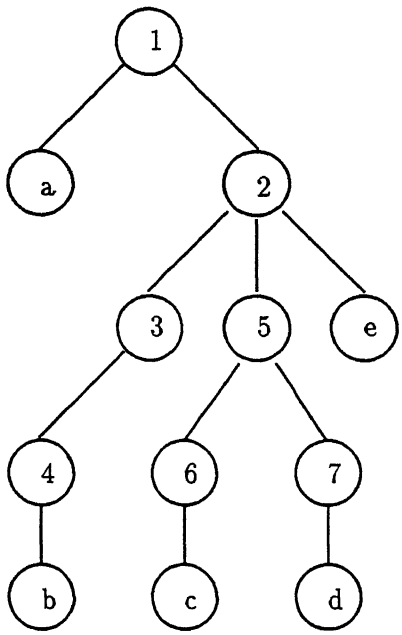

Figure 3.2: An arbitrarily nested loop and its tree representation

allel execution time of a DOALL loop L is given by

$$T_p(B_L) = \left\lceil \frac{N}{p} \right\rceil * b$$

where b is the serial execution time of the loop body.

Theorem 3.1 *Parallel execution time of a* DOACR *loop with N iterations, a delay of d and a loop body of size b that executes on a system with p processors is given by*

$$T_p(b) = (\left\lceil \frac{N}{p} \right\rceil - 1) * \max(b, pd) + d * ((N-1) \bmod p) + b \qquad (3.1)$$

Proof : Refer [15] ■

Now consider the problem of assigning p processors to a perfectly nested loop of nest depth 2. A possible assignment is to partition the p processors into q clusters where each cluster contains $r = \lfloor p/q \rfloor$ processors and then assign q clusters to the outer loop and r processors of each cluster to the inner loop. Under this assignment, inner loop will be executed as if only r processors are available. Since q clusters are assigned to the outer loop, at most q iterations of the outer loop can be executed simultaneously. This appears as if the outer loop is being executed with only *q processors.* Henceforth, the term *processor* is used generically and refers to clusters of physical processors.

Execution of the inner loop will now take $T_{inner}(b_{inner})$ units of time and that of the outer loop takes $T_{outer}(b_{outer})$ units of time. Since each execution of the loop body of the outer loop is the execution of the inner loop, we have $b_{outer} = T_{inner}(b_{inner})$. Hence, parallel execution time of the above processor assignment should be $T_{outer}(T_{inner}(b_{inner}))$. Generalization of this idea is found in the algorithm **OPTAL** [15].

Based on the dynamic programming technique, the algorithm OPTAL finds the optimal processor assignment. An assignment function $G_i^j(q)$ is defined to be the parallel execution time of an optimal assignment of q processors to loop j at level i. A $\lambda \times p$ table is used to store the intermediate values of $G_i^j(q)$. The algorithm works in two steps. In the first step, parallel execution times of all loops at level k of the tree (where k is the nest depth) is computed as follows.

$$G_k^i(q) = T_q^i(b_i) \; ; \; q = 1, 2, \cdots p$$

where $T_q^i(*)$ is given by equation 3.1.

The second step is defined recursively. Parallel execution time of the optimal assignment of q processors to loop j in levels i through k $(i < k)$ (assuming optimal assignment of q processors to loops at level $i+1$ is known) is then

computed by

$$G_i^j(q) = \min_{1 \le r \le q} \{T_r^j(\sum_{n\, child\, of\, j} G_{i+1}^n(\left\lceil \frac{q}{r} \right\rceil))\} \tag{3.2}$$

Equation 3.2 is computed for all nodes(loops) at level i. The summation in 3.2 accounts for all the loops that are nested in the loop j. Parallel execution time of the optimal assignment of p processors to the loop nest L is given by $G_1^L(p)$.

A detailed algorithm is given in the figure 3.3.

Algorithm **OPTAL** :

Input : A loop L of nest depth k and p processors

Output : An optimal processor allocation profile of p to L

Method :

Step 1 : **For** all loops j at level k of the loop tree
  For all $q = 1, 2, \cdots p$, compute
    $G_k^j(q) = T_q^j(B_j)$

Step 2 : **For** $i = k - 1$ **downto** 1
  For all loops j at level i
    For all $q = 1, 2, \cdots p$, compute
    $G_i^j(q) = \min\{T_r^j(\sum_{n\, child\, of\, j} G_{i+1}^n(\lfloor q/r \rfloor))/r = 1, 2, \cdots q\}$
  and store the results in the j-th row of the table T.

Step 3 : Output the processor allocation profile computed by $G_1^L(p)$.

Figure 3.3: algorithm **OPTAL**

OPTAL automatically constructs the detailed processor assignment vector. For each loop j at level i, the optimal number of processors assigned to that loop when q processors are available corresponds to the minimum term in equation 3.2 and the number of processors assigned to loops nested in loop j is $\lfloor q/r \rfloor$. Hence,

optimal processor assignment is constructed recursively.

Lemma 3.1 *Time complexity of the algorithm OPTAL is $O(\lambda p^2)$ where λ is the number of individual loops in the nested loop and p is the number of processors.*

Proof : In step 2 of OPTAL, the assignment function is computed p times for each loop in the nest. Also each evaluation of the assignment function involves finding the minimum of at most p terms. Therefore complexity of evaluating G is $O(p^2)$ without counting the additions. Since there are λ loops in the nest, the overall time complexity of OPTAL should be $O(\lambda p^2)$. Time complexity of OPTAL can be reduced to $O(\lambda p \log p)$ by executing OPTAL itself in parallel on p processors. ■

OPTAL also computes all optimal assignments of $1, 2, \cdots p-1$ processors to an arbitrarily nested loop as intermediate results during the computation of the optimal assignment of p processors to the loop nest. In this sense, OPTAL is complete. OPTAL can hence be used to find the maximum number of useful processors. A number of processors p is said to be *useful* with respect to a loop nest if there exists an assignment which can assign exactly p processors to the loops in the nest. Given p processors and a loop nest L, maximum number of useful processors is the minimum q ($q \leq p$) such that the parallel execution time of L with q processors is never worse than that of L with p processors. OPTAL can easily find this out.

Incomplete algorithms on the other hand, compute only those assignments that are useful in determining the optimal assignment of p processors to the loop nest and hence tend to be faster than complete algorithms. If the number of available processors is fixed, then it would be more efficient to use incomplete algorithms to generate allocation profiles.

As seen earlier, computing time spent in step 2 of OPTAL is $O(p^2)$. However

the number of all possible assignments of p processors, denoted by $A_q = \{(r,s)/1 \leq r * s \leq p\}$, is found to be $O(p \log p)$.

$$|A_q| = \sum_{r=1}^{p} \left\lfloor \frac{p}{r} \right\rfloor \leq \sum_{r=1}^{p} \frac{p}{r} = p * \sum_{r=1}^{p} \frac{1}{r} = O(p \log p)$$

This indicates that OPTAL computes each assignment $O(p/\log p)$ times on an average. An improvement in this direction is found in the algorithm **SECAL** [17]. SECAL computes the optimal assignment of p processors to any arbitrarily nested loop in $O(\lambda p \log p)$ time. Time complexity of SECAL can further be reduced to $O(\lambda \log p)$ by executing SECAL itself in parallel on p processors.

Wang and Wang[17] have also proposed an incomplete algorithm **FINAL** which can compute optimal assignment of p processors to a nested loop in $O(\lambda p^{3/4})$ time.

As stated earlier, static allocation schemes sound attractive as long as the iterations have no variance in their execution times and the loop bounds are known at compile time. Unfortunately, this is not the case in most of the practical programs and the schedules generated by the static schemes can be far from optimal. Since at run time we have more information about the program, we could perform processor assignment just before the execution begins using these algorithms. However, the overheads associated with these methods could be prohibitively large. It is yet to be seen whether fast incomplete algorithms can perform as good as any traditional dynamic scheduling schemes. In general, static allocation schemes may not find as wide an application as that of dynamic scheduling strategies.

3.5 Transformations

Scheduling techniques can become effective if certain transformations are performed on these parallel loops. It has been observed that in most cases, execu-

tion time is improved after applying these transformations. A lot of techniques that transform a parallel program into a semantically equivalent one can be found in literature. Some commonly used transformation techniques which help in improving the performance of scheduling algorithms are loop coalescing [15], loop interchange [18], loop distribution [18].

3.5.1 Loop Coalescing

Loop coalescing transforms a perfectly nested loop into a single loop. This technique expresses all indices in the original loop as functions of a single index of the transformed loop as shown in figure 3.4. Note that loop index is a shared variable and all accesses to it are by necessity synchronized. This is a time consuming process and can greatly deteriorate the performance. Since loop coalescing transforms a nested loop into a single loop, we now need to perform synchronized access to a single loop index. Clearly, this is the minimum synchronization overhead we can achieve with any scheduling scheme.

$$
\begin{array}{lcl}
\texttt{DO } I_1 = 1,N_1 & & \texttt{DO } I = 1,N_1*N_2 \\
\quad \texttt{DO } I_2 = 1,N_2 & & \quad I_1 = \lceil I/N_2 \rceil \\
\quad\quad \{B\} & \Longrightarrow & \quad I_2 = I - N_2 * \lfloor (I-1)/N_2 \rfloor \\
\quad \texttt{ENDDO} & & \quad \{B\} \\
\texttt{ENDDO} & & \texttt{ENDDO}
\end{array}
$$

Figure 3.4: Loop coalescing

If $p = p_1 * p_2 * \cdots * p_k$ processors are assigned to a perfectly nested loop of nest depth k such that p_i processors ($i = 1, 2, \cdots k$) are allocated to loop i of the nest, then parallel execution time is given by $\prod_{i=1}^{k} \lceil N_i/p_i \rceil$. If on the other hand, coalescing is performed before processor assignment, then parallel

execution time will be $\lceil \prod_{i=1}^{k} N_i/p_i \rceil$.

$$\left\lceil \prod_{i=1}^{k} \frac{N_i}{p_i} \right\rceil \leq \prod_{i=1}^{k} \left\lceil \frac{N_i}{p_i} \right\rceil$$

3.5.2 Loop Interchange

The process becomes bit complicated in case of hybrid loops. Consider for example, a perfectly nested loop of nest depth 3 (see figure 3.5). The loop at nest

```
DOALL I1 = 1,N1                          DO I2 = 1,N2
    DO I2 = 1,N2                            DOALL I1 = 1,N1
        DOALL I3 = 1,N3                         DOALL I3 = 1,N3
            {B}                   ==>               {B}
        ENDDOALL                                ENDDOALL
    ENDDO                                   ENDDOALL
ENDDOALL                                 ENDDO
                                              ||
                                              \/
                                   DO I2 = 1,N2
                                      DOALL I = 1,N1*N3
                                          . . .
                                          {B}
                                      ENDDOALL
                                   ENDDO
```

Figure 3.5: Loop Interchange followed by Coalescing

level 2 is a serial loop. If dynamic scheduling is employed, then at most N_3 iterations can be scheduled simultaneously. The presence of the serial loop in the middle limits the extent to which parallelism can be exploited. Suppose we permute the indices of the outer two loops, a technique called loop interchange, and subsequently perform loop coalescing, then we can schedule $N_1 * N_3$ iterations at a time. Loop interchange cannot be performed in all cases. The following lemma due to Polychronopoulos[15] states when loop interchange can be performed.

Lemma 3.2 *In a hybrid perfectly nested loop, it is legal to interchange any* DOALL *loop with any serial* DO *or parallel* DOACR *loop that is at a deeper nest level.*

Proof : Refer [15]. ■

It is a common practice to repeatedly perform loop interchange in a nested loop so that all the DOALL loops are pushed to the deepest nest levels possible and then coalesce adjacent DOALL loops before scheduling them.

3.5.3 Loop Distribution

Loop distribution converts an m-way nested loop into a sequence of perfectly nested loops which are easier to deal with. If there are no dependences between the statements in the loop body, then this technique distributes the loop to all these statements (see figure 3.6). When coalescing is applied in conjunction with loop distribution, multi-way nested loops can be transformed into single loops.

```
DOALL I1 = 1,N1                          DOALL I1 = 1,N1
      DOALL I2 = 1,N2                          DOALL I2 = 1,N2
            {B1}                                     {B1}
      ENDDOALL                                 ENDDOALL
      DOALL I3 = 1,N3              ==>   ENDDOALL
            {B2}                         DOALL I1 = 1,N1
      ENDDOALL                                 DOALL I3 = 1,N3
ENDDOALL                                             {B2}
                                               ENDDOALL
                                         ENDDOALL
```

Figure 3.6: Loop distribution

Chapter 4

Mapping onto the underlying Architecture

In the previous chapters, we looked at some techniques for partitioning a program and then scheduling it. In this chapter we discuss some aspects of obtaining a mapping of the restructured program to the underlying architecture. We attempt to translate a parallel program to which scheduling decisions have been made deterministically (as seen in earlier chapters) into a single program multiple data (**SPMD**) version [8]. This SPMD program executes directly on all the nodes of the multiprocessor. However, each processor executes only those portions of the program assigned to it. As seen earlier, synchronization between the processing elements is necessary to ensure semantic equivalence. We try to generate synchronization instructions automatically wherever required. We can either go on to generate machine code or translate the program back into a high level code which can be easily read by an interested user. We choose to take the second approach and present the user with these details in case he is eager to get more from the system (Refer figure 1.2 of chapter 1).

4.1 Assumptions

Before proceeding to the actual process of translation, we need to make some assumptions about the underlying architecture. We assume a shared memory machine model where we have identical parallel processors which share a common address space (explained in the first chapter). Each processor is scalar and communication between processors takes place through the shared memory. Each processor is identified by a unique number. For example, if there are p processors in the system, then each processor is identified by a number lying between 0 and $p-1$. We call this the *processor identifier* $\$\_Pid$. We also assume that the applications are run in batch mode. We consider only non-preemptive schedules.

Since loops form the main target of restructuring and subsequent scheduling, we continue to focus our attention on translating parallel loops correctly. The next section addresses the problem of how to obtain a SPMD version of an arbitrarily nested loop systematically.

4.2 Handling parallel loops

We are given an arbitrarily nested loop and a processor assignment profile to it. The problem is to determine which processor is going to execute a particular iteration instance. To get the flavor of the problem at hand, consider a perfectly nested loop of depth 2 which is to be executed on a 4 processor machine. Assume an allocation vector which assigns 2 processors to the outer loop and 2 processors to the inner. A possible way of executing this loop nest on the 4 processors is as shown in figure 4.1.

In other words, we have to determine the new iteration bounds of the individual loops in the nest for each processor involved in executing it. i.e. we

```
DOALL I = 1,10
      DOALL J = 1,10
            { B}
      ENDDOALL
ENDDOALL
```

processor 0	processor 1	processor 2	processor 3
DO I = 1,5	DO I = 1,5	DO I = 6,10	DO I = 6,10
DO J = 1,5	DO J = 6,10	DO J = 1,5	DO J = 6,10
{ B }	{ B }	{ B }	{ B }
ENDDO	ENDDO	ENDDO	ENDDO
ENDDO	ENDDO	ENDDO	ENDDO

Figure 4.1: An example loop nest executed on 4 processors

have to find general expressions for the values taken by each of the iteration variables in terms of the processor identifiers. Note that since we are generating a SPMD program, the loop indices are now local to each processor. We take the approach of allocating a block of successive iterations of each loop in the nest to a particular processor.

Let us begin with a simple example of a perfectly nested loop of depth 2 (see figure 4.2(a)). The figure shows a one-way nested loop of depth 2. This loop nest is to be executed on a p-processor system. Let the processors be numbered from 0 to $p-1$. That is their $\$\_Pid$'s range from 0 to $p-1$. Further, assume that a processor assignment is made such that p_1 processors are allocated to the outer loop and p_2 processors to the inner, where $p_1 * p_2 \leq p$. For simplicity, we assume that $p_1 * p_2 = p$. The above assignment has partitioned the p processors into p_1 clusters of p_2 proccessors each. With this assignment, the loop nest will be executed as if p_1 processors are available to the outer loop and p_2 processors are available to the inner. Note that there are a total of $N_1 * N_2$ iterations

and each processor has to execute a maximum of $\lceil N_1/p_1 \rceil * \lceil N_2/p_2 \rceil$ iterations. N_2 iterations of the inner loop are distributed among p_2 processors of every cluster. Each processor gets $\lceil N_2/p_2 \rceil$ iterations of the inner loop. The first chunk of $\lceil N_2/p_2 \rceil$ iterations is executed by the first processor of every cluster, the second chunk by the second processor of every cluster and so on. i.e. processors $0, p_2, 2p_2, \cdots (p_1 - 1) * p_2$ will execute the first $\lceil N_2/p_2 \rceil$ iterations of the inner loop and so on. In general, a processor i would execute the following iterations of the inner loop.

$$(i \bmod p_2) * \left\lceil \frac{N_2}{p_2} \right\rceil + 1, \cdots ((i \bmod p_2) + 1) * \left\lceil \frac{N_2}{p_2} \right\rceil$$

Since the outer loop will be executed as if only p_1 processors are available, each processor is going to get $\lceil N_1/p_1 \rceil$ iterations of the outer loop. This amounts to saying that cluster 0 executes the iterations $1, 2, \cdots \lceil N_1/p_1 \rceil$, cluster 2 executes the iterations $\lceil N_1/p_1 \rceil + 1, \lceil N_1/p_1 \rceil + 2, \cdots 2\lceil N_1/p_1 \rceil$ and so on. A processor i belongs to a cluster j if $\lfloor i/p_2 \rfloor = j$. Hence a processor i will execute the following iterations of the outer loop.

$$(\left\lfloor \frac{i}{p_2} \right\rfloor) * \left\lceil \frac{N_1}{p_1} \right\rceil + 1, \cdots (\left\lfloor \frac{i}{p_2} \right\rfloor + 1) * \left\lceil \frac{N_1}{p_1} \right\rceil$$

Therefore, when we translate the program into SPMD code the above loop is transformed as shown in figure 4.2(b).

One can easily extend this argument to perfectly nested loops of any depth k. We only have to find the constants $(C_1, C_2, \cdots C_k)$ such that each of the loop indices I_i takes the values $C_i * \lceil N_i/p_i \rceil + 1, \cdots (C_i + 1) * \lceil N_i/p_i \rceil$. We have the following result.

Theorem 4.1 *If $(p_1, p_2, \cdots p_k)$ is a processor assignment to a perfectly nested loop of depth k, $L(N_1, N_2, \cdots N_k)$ (where $I_1, I_2, \cdots I_k$ are the loop indices) such that p_i processors are allocated to the loop at depth i, then each processor is going to execute the following iterations of the loop at depth m, $1 \leq m \leq k$.*

```
DOALL I_1 = 1,N_1
      DOALL I_2 = 1,N_2
            {B}
      ENDDOALL
ENDDOALL
```

$$\begin{array}{l}
\texttt{DO } I_1 = (\lfloor \$_Pid/p_2 \rfloor) * \lceil N_1/p_1 \rceil + 1, (\lfloor \$_Pid/p_2 \rfloor + 1) * \lceil N_1/p_1 \rceil \\
\quad \texttt{DO } I_2 = (\$_Pid \bmod p_2) * \lceil N_1/p_1 \rceil + 1, (\$_Pid \bmod p_2 + 1) * \lceil N_1/p_1 \rceil \\
\qquad \{B\} \\
\quad \texttt{ENDDO} \\
\texttt{ENDDO}
\end{array}$$

Figure 4.2: A DOALL loop nest and its SPMD version

$$\begin{aligned}
C_m &= \left\lfloor \frac{\$_Pid \bmod \prod_{i=m}^{k} p_i}{\prod_{i=m+1}^{k} p_i} \right\rfloor \\
I_m &= C_m * \left\lceil \frac{N_m}{p_m} \right\rceil + 1, \cdots (C_m + 1) * \left\lceil \frac{N_m}{p_m} \right\rceil
\end{aligned} \tag{4.1}$$

Proof : We prove this by induction. One can easily see that 4.1 holds good for nested loops of depths 2 and 1. Let us assume that 4.1 holds good for any loop nest of depth m.

for $i = 1, 2, \cdots m$

$$\begin{aligned}
C_i &= \left\lfloor \frac{\$_Pid \bmod \prod_{j=i}^{m} p_j}{\prod_{j=i+1}^{m} p_j} \right\rfloor \\
I_i &= C_i * \left\lceil \frac{N_i}{p_i} \right\rceil + 1, \cdots (C_i + 1) * \left\lceil \frac{N_i}{P_i} \right\rceil
\end{aligned}$$

We have to prove that 4.1 holds for loop nests of depth $m+1$ also. Consider a loop nest of depth $m+1$. Clearly, p_1 processors are allocated to the outermost loop while $\prod_{i=2}^{m+1} p_i$ processors are assigned to the inner loop nest of depth m. That is, we now have p_1 clusters of $\prod_{i=2}^{m+1} p_i$ processors each. The first cluster

will now execute the first $\lceil N_1/p_1 \rceil$ iterations of the outermost loop, the second cluster will execute the second $\lceil N_1/p_1 \rceil$ iterations and so on. A processor(\$_*Pid*) will belong to a cluster j if $\lfloor \$\_Pid/\prod_{i=2}^{m+1} p_i \rfloor = j$. Hence a processor(\$_*Pid*) will execute the following iterations of the outermost loop.

$$I_1 = \left\lfloor \frac{\$\_Pid}{\prod_{i=2}^{m+1} p_i} \right\rfloor * \left\lceil \frac{N_1}{p_1} \right\rceil + 1, \cdots (\left\lfloor \frac{\$\_Pid}{\prod_{i=2}^{m+1} p_i} \right\rfloor + 1) * \left\lceil \frac{N_1}{p_1} \right\rceil$$

Clearly, \$_*Pid* can be written as \$_*Pid* mod $\prod_{i=1}^{m+1} p_i$ (because \$_*Pid* is always less than $\prod_{i=1}^{m+1} p_i$).

Therefore,

$$\begin{aligned} I_1 &= C_1 * \left\lceil \frac{N_1}{p_1} \right\rceil + 1, \cdots (C_1 + 1) * \left\lceil \frac{N_1}{p_1} \right\rceil \\ where\ C_1 &= \left\lfloor \frac{\$\_Pid \bmod \prod_{i=1}^{m+1} p_i}{\prod_{i=2}^{m+1} p_i} \right\rfloor \end{aligned}$$

Note that by assumption, 4.1 holds for the sub-nest at nest level 2 for it is a loop nest of depth m.

Hence the result. ∎

We can transform arbitrarily nested loops also using the same argument. The following example (see figure 4.3) shows how translation can be done for arbitrarily nested loops. This is possible only when the total number of processors allocated to all loops at the same nest level is the same. This is exactly the situation when OPTAL is used to generate processor assignments.

4.3 Synchronization

More often, the processors involved in executing a loop nest start executing it at different instants of time. Also, loops in general have bodies with variable execution times. Hence one would expect the processors executing a loop nest to finish

```
DOALL I = 1,N1
      DOALL J = 1,N2
            { B1 }
      ENDDOALL
      DOALL k = 1,N3
            { B2 }
      ENDDOALL
ENDDOALL
```

(a)

DO I = $\lfloor \$\_Pid/p_2 \rfloor * \lceil N_1/p_1 \rceil + 1, (\lfloor \$\_Pid/p_2 \rfloor + 1) * \lceil N_1/p_1 \rceil$
   DO J = $(\$\_Pid \bmod p_2) * \lceil N_2/p_2 \rceil + 1, ((\$\_Pid \bmod p_2) + 1) * \lceil N_2/p_2 \rceil$
      { B1}
   ENDDO
   DO K = $(\$\_Pid \bmod p_2) * \lceil N_3/p_2 \rceil + 1, ((\$\_Pid \bmod p_2) + 1) * \lceil N_3/p_2 \rceil$
      { B2}
   ENDDO
ENDDO

(b)

Figure 4.3: An arbitrarily nested DOALL loop and its SPMD version

at different times. If the succeeding tasks depend on this task(loop nest), then the processors which finish executing their part of the loop nest earlier cannot proceed on to other tasks. They have to idly wait for others to finish executing the loop. Based on these observations, we conservatively set a barrier at the end of every DOALL loop. However, by naively setting a barrier at the end of every parallel loop, one may actually perform some unnecessary synchronizations. In perfectly nested DOALLs, for example, only one synchronization is sufficient at the end of the entire loop nest. Since setting barriers introduces significant run

time overheads, they should be used economically. The processors involved in executing the loop nest also have to synchronize with those executing the tasks on which the loop under consideration depends. The loop should be executed only after all the dependence constraints are satisfied. Hence we set a barrier at the beginning of every parallel loop nest.

4.4 Handling other constructs

So much about the parallel loops. Let us now divert our attention to program constructs other than parallel loops. An implicit assumption made is that all tasks other than parallel loops are strictly sequential. By putting this restriction, we may lose some parallelism that could be present in a piece of straight line code. The justification given is such parallelism cannot be used to advantage. Exploiting this parallelism may finally end up in an execution time worse than that of serial execution. This happens because of the unavoidable scheduling overheads. Exploiting statement level parallelism hence makes sense only when the task size is large enough to amortize the overheads suffered during scheduling. We may, however, exploit this statement level parallelism within a processor, if it allows for pipelining. This problem is not discussed because it requires the knowledge of the processor hardware.

4.5 Prefetching

So far we have discussed mapping of parallel programs to shared memory multiprocessors assuming that the individual processors have negligible local memory. In commercial systems however, processors come with substantial amount of private memory. This memory hierarchy can be used to advantage in reducing large memory access times. Accessing the shared memory involves entering the

critical sections, which is a time consuming process. We could probably avoid accessing shared memory frequently by fetching a block of data that could be used in the near future, into the local memory. This technique is popularly referred to as *prefetching.*

Prefetching can be formally defined as the process of transferring a block of data asynchronously from the global memory to local memory before the computation that uses this data [7]. We are only concerned with array references that can be prefetched before the execution of loops. Consider the following example where prefetching is done before the loop.

```
prefetch B(1..N)
DO I = 1,N
   A(I) = B(I) + 1
ENDDO
```

In the above example, all references to the array B are local inside the loop. Thus by prefetching data, we have been able to reduce the the memory access time.

Amount of data to be prefetched depends on the point at which data is prefetched. In the following example (see figure 4.4), we can pull out the reference to B from both the loops or we can pull out the reference from the inner loop only.

In the former case, we make one prefetch of $N * M$ elements. In the latter, we generate N prefetches of M elements each. The disadvantage with the first approach is the size of the local memory required. The second approach requires less local memory but fetching overheads are incurred N times. We feel that if a system has processors with substantial amount of local memory, then it would

```
DO I = 1,N
   DO J = 1,M
      A(I,J) = B(I,J)
   ENDDO
ENDDO
```

```
prefetch B(1..N,1..M)
DO I = 1,N
   DO J = 1,M
      A(I,J) = B(I,J)
   ENDDO
ENDDO
```

```
DO I = 1,N
   prefetch B(I,1..M)
   DO J = 1,M
      A(I,J) = B(I,J)
   ENDDO
ENDDO
```

Figure 4.4: An example of Prefetching

be wise to pull out the references to array elements from the entire loop nest.

Many a time, dependences cause data to be prefetched unnecessarily. Consider the following examples.

```
prefetch B(1..10)
DO I = 1,10
   IF I < 5 THEN
      A(I) = B(I) + 1
ENDDO
```
(a)

```
prefetch A(1..10)
DO I = 1,10
   A(I) = I
   C(I) = A(I)+1
ENDDO
```
(b)

Figure 4.5: Cases when prefetching is suppressed

If we pull out the reference to B from the loop (figure 4.5(a)), we would be doing some unnecessary work. Here we are fetching 10 elements of B, but only 5 elements are being used. By fetching too much data that will not be used we might actually increase the program execution time instead of decreasing it. In figure 4.5(b), the prefetched data would be modified even before it is used. This could even result in incorrect execution because of inconsistency in data.

In such cases we avoid fetching the data. In general, a prefetch is suppressed when there is a possibility of either

- prefetching data that might not be used or
- some of the prefetched data being modified before it is used. This implies

that the data to be prefetched must be 'read only' between the time it is prefetched and the time it is used.

Consider the following example (see figure 4.6).

```
DO I = 1,N
   A(2I) = ...
   ...  = ...  A(2I+1) ...
ENDDO
```

Figure 4.6: Example where prefetching is possible but suppressed

Here, even though the array A is being modified inside the loop, the elements which are to be read are not written into. We could perform prefetching here. However this needs sophisticated analysis very similar to the traditional dependence analysis. We don't see much improvement by performing such analysis and don't attempt to do so.

4.6 Data Distribution

Many researchers ([6], [8]) feel that maximum benefit of multiprocessing can be obtained by cleverly distributing the data among the processing elements. When data required for some computation resides in the global memory or in the local memory of some other processor, undesired delays are introduced either due to serialization of concurrent requests from multiple processors or due to the wait times arising from the message passing primitives. The problem is more serious in case of distributed memory systems. More frequently, the compiler fails to identify an efficient data distribution pattern automatically. It hence becomes the responsibility of the user to specify the right data distribution pattern for his application. Unfortunately, standards to specify data distribution patterns

have not yet evolved. Worse yet is the fact that parallel languages are still in the devoloping stages. Fortran D, a parallel dialect of Fortran 77, has been recently developed which supports data distributions [8]. Fortran D provides rich support to specify data distributions. We do not get into the details of how to specify data distributions in Fortran D. An interested reader is referred to the Appendix.

Without limiting ourselves to Fortran D, we specify a data distribution as shown.

```
INTEGER MATRIX(100)
DISTRIBUTE MATRIX(4)
```

This means that MATRIX is an array of 100 elements distributed on 4 processors. Now these 4 processors have 25 elements of the array MATRIX. Which 25 elements of MATRIX go to a particular processor can also be specified in Fortran D. We however, do not go into these details and assume that MATRIX is distributed in such a way that each processor gets a block of 25 successive elements. We assume that only arrays are distributed. Note that arrays can be distributed in all its dimensions. An array if not distributed, is assumed to reside in the global memory.

```
INTEGER MATRIX(100,100)
DISTRIBUTE MATRIX(4,4)
```

In this example, MATRIX is distributed among 16 processors in such a way that each processor gets 25 X 25 elements of MATRIX. Now we have to assign tasks to the individual processors so that the interprocessor communication is minimized. In other words, we have to assign tasks to processors in such a way that

```
INTEGER MATRIX(100)
DISTRIBUTE MATRIX(4)

DOALL I = 1,100
     MATRIX(I) = ...
ENDDOALL
```

Figure 4.7: An example of data distribution

the assigned tasks operate only on data local to that processor. We follow the principle that "*A processor can modify data only if it owns them*", usually referred to as the 'owner computes' rule. We deal with the problem of partitioning parallel loops only.

Consider a `DOALL` loop which initializes the elements of a *distributed* array `MATRIX` (see figure 4.7). It would be desirable to assign a processor those iterations only which access elements of `MATRIX` local to it.

We need to change the loop bounds suitably, so that only local data is accessed as far as possible. In general, the array indices can be complex expressions of the loop indices and calculating the new bounds for the loops becomes difficult. We take an easy way out. Instead of changing the loop bounds, we allow the processors to go through all the iterations. The loop bodies however will be executed conditionally. Every write to a distributed array location is preceded by a check for its locality. A write takes place only if the location to be written into is not remote (see figure 4.8).

In the above example, even though each processor executes all the 100 iterations, the loop body is executed only 25 times. Note that we have added an extra check for every assignment statement in the loop body. This approach will give faster executions than their sequential counterparts only if the cost of

```
INTEGER MATRIX(100)
DISTRIBUTE MATRIX(4)

DOALL I = 1,100
      A(I) = B(I) + C(I)
ENDDOALL
```

```
INTEGER MATRIX
DISTRIBUTE MATRIX(4)

DO I = 1,100
   IF A(I) local
      A(I) = B(I) + C(I)
ENDDO
```

Figure 4.8: A data distributed program and its SPMD equivalent

performing a check is much less than that of the assignment statements. This is in fact the case with many numerical programs. We can make these checks faster by performing them in the hardware.

The problem that remains is given a data distribution, how to determine which array location is local (or remote) to a particular processor. Again we try to find a general expression to the bounds of the array in each dimension as functions of the processor identifier.

Let DISTRIBUTE $A(p_1, p_2, \cdots p_k)$ be a data distribution of the array $A(N_1, N_2, \cdots N_k)$. An array reference $A(I_1, I_2, \cdots I_k)$ is local to a processor p if $I_i (i = 1, 2, \cdots k)$ satisfies the following relation.

$$\left\lfloor \frac{p \bmod \prod_{j=i}^{k} p_j}{\prod_{j=i+1}^{k} p_j} \right\rfloor * \left\lceil \frac{N_i}{p_i} \right\rceil + 1 \leq I_i \leq \left(\left\lfloor \frac{p \bmod \prod_{j=i}^{k} p_j}{\prod_{j=i+1}^{k} p_j} \right\rfloor + 1 \right) * \left\lceil \frac{N_i}{p_i} \right\rceil \tag{4.2}$$

Equation 4.2 follows from a simple induction over i, on the same lines as that of the theorem 4.1.

Prefetching also needs to be performed if data is distributed. Prefetching should be performed only if the time spent in fetching data is compensated by the speedup obtained in the execution of the loop.

Chapter 5

Implementation

This chapter gives a brief picture of the implementation of some of the concepts discussed so far. We present only those issues which could appeal to an implementor. Note that the emphasis is on exploiting parallelism in loops. We now consider some features of the input language, structure of the intermediate representation and the data structures used, and the policies adopted.

5.1 Input Language

Since the underlying architecture assumed was a shared memory multiprocessor, we needed a language which allows the user to specify the data distributions efficiently. Fortran D ideally suited our needs. Details of how to specify data distributions in Fortran D can be found in Appendix. We do not stick to Fortran D syntax strictly. We bypass the problem of aligning arrays with respect to decompositions by not considering the decompositions. In other words, every array to be distributed is aligned with a decomposition that is a mirror image of itself. This alignment being redundant in this context, is neglected. We do not allow the user to specify how the array elements should be distributed on the different processors. We implicitly assume that the arrays are distributed

'blockwise'. The user can specify the number of processors to be assigned to each dimension as done in Fortran D. The following example makes the idea clear.

```
INTEGER MATRIX(100)
DISTRIBUTE MATRIX(4)
```

This means that the array MATRIX is distributed among 4 processors blockwise. i.e. the first processor gets first 25 elements of MATRIX, the second processor gets the next 25 elements of MATRIX and so on. Arrays can be distributed in all dimensions. A single processor is assigned to a dimension if it is not distributed.

```
INTEGER MATRIX(100,100)
DISTRIBUTE MATRIX(4,2)
```

Here MATRIX is being distributed among 8 processors such that the rows of MATRIX are spread among 4 processors while the columns are spread among 2 processors. Each processor now will have MATRIX(25,50). Which 25 rows and which 50 columns are in a particular processor is as shown.

Processor	0	1	2	3	4	5	6	7
Rows	1..25	1..25	26..50	26..50	51..75	51..75	76..100	76..100
Columns	1..50	51..100	1..50	51..100	1..50	51..100	1..50	51..100

Table II: Distribution of MATRIX(100,100) on 8 processors

We do not allow dynamic data decompositions unlike Fortran D. That is, data decompositions are specified only once in the beginning of the program and do not change throughout the execution of the program. In other words, DISTRIBUTEs are interpreted as declarations rather than executable statements.

The rest of the language is very similar to Fortran 77 except for the FORALL construct. DOALL loops (explained earlier) can be specified using the FORALL construct of Fortran D. Syntax of the FORALL construct is the same as that of the DO loop as shown below. Fortran D does not support DOACR type of loop constructs.

```
forall I = 1,N
       { B }
endfor
```

Two major steps in writing a parallel program are selecting a proper data distribution pattern and using it to derive node programs with explicit data movement. We leave the job of specifying data distributions to the user. We try to generate node programs with explicit data transfers for a given data distribution. The main objective is to exploit parallelism and reduce the communication costs. As explained in the previous chapter, we translate the parallel program into a SPMD program in a Fortran like language.

We also provide tools which enable the user to interact with the compiler. Often, loop bounds are not known at compile time and the compiler fails to estimate the execution times of such constructs accurately. This may result in inefficient schedules. In such cases, the user can provide some useful information regarding the loop bounds to the compiler through *directives*. The directives supported in FRAMES have been modeled on those supported by the optimizing Fortran compiler for *Convex C-220* [5] and they have the following syntax.

C$DIR <directive> parameter(optional)

More about directives can be found in [3]. User can specify the maximum number of iterations a loop can have, through the max_trips directive. The max_trips directive has the following format.

C$DIR max_trips *count*

where *count* indicates the maximum number of iterations the loop following this directive can have.

Once the compiler gets this information, it can make a rough estimate of the execution times of the loops and obtain near optimal schedules.

Some useful directives that can be included are as follows.

```
C$DIR begin_task
statements
C$DIR end_task
```

Since we consider all tasks other than loops to be sequential, we may lose some parallelism that could be present in these 'sequential' tasks. If the user feels that this parallelism can be exploited to advantage, then he could direct the compiler to break this single sequential task into multiple tasks so that they can be executed in parallel.

```
C$DIR par_begin
tasks
C$DIR par_end
```

This directive informs the compiler that the tasks within `par_begin` and `par_end` can be executed simultaneously. Each of these tasks could be either serial or parallel. Should the nesting of these directives be allowed or not is again an interesting issue.

5.2 The Intermediate representation(IR) used

A suitable data structure is needed which can effectively encapsulate all the necessary information in the program. The data structure should be easy to manipulate during the various restructuring and scheduling stages. An abstract syntax tree ideally suited our needs. Each node in the abstract syntax tree is as shown in Figure 5.1.

Every node is a syntactic unit and contains all data related to it. For example, with respect to a loop, information about the index variable, lower and upper bounds of the loop, the stride and the loop body is contained in the node representing it. Most of the fields are self explanatory. More about the IR can be found in Ananda[3], Rajeev[13].

Fields of interest are the `assign_cluster` and `assign_processors`. Recall that when processor assignment is performed to loop nests, one needs to know how many processors/clusters of processors are assigned to each individual loop in the nest. The two fields respectively contain the information about the number of processors and clusters assigned. Though we could have done away with one of the fields (`assign_cluster` and `assign_processors` are inter-related.), we choose to retain both of them as it makes the job of programming easier.

5.3 The Parser

We have built a parser which accepts a program written in Fortran D. Though issues such as error recovery and handling complex data types (records, files etc.) are not addressed, the parser meets the objective of obtaining an adequate IR. As stated earlier the IR is an abstract syntax tree. The grammar for the language accepted by the parser can be found in Ananda[3]. It suffices to state here that the front end (refer figure 1.2) generates a control flow graph (CFG)

```
node {
   int node_type;   /* discriminator for type of node */
   union {
      int   i_val;  /* integer value to be stored */
      float f_val;  /* floating point value to be stored */
      char  *s;     /* pointer to a character string */
      struct node *node_ptr ; /* pointer to another node */
   } elements[5];
   struct node    *next;    /* pointer to the next node */
   struct id_data *table_entry_ptr; /* pointer to symbol table */
   int            misc;             /* any other misc data */
   int            label;            /* label of the statement */
   int            statno;           /* unique statement number */

   /* control flow graph related fields */

   struct node   *t_edge;           /* true edge */
   struct node   *f_edge;           /* false edge */
   struct node   *u_edge;           /* unconditional edge */

   /* data flow analysis related fields */

   int            df_entry;
   int            visited;

   /* fields related to scheduling */

   int          assign_cluster;    /* clusters assigned */
   int          assign_processors; /* total processors assigned */
      };
```

Figure 5.1: Structure of each node in the abstract syntax tree

which is passed to the subsequent stages for further analysis.

5.4 The Task dependence graph

The task dependence graph represents the dependence relations between the different tasks in the program. The scheduler then examines every node in the task graph and schedules it depending on its nature. Every task is identified by a unique number. Associated with each task is a number that represents the layer to which it belongs (see chapter 2). Based on the control flow (obtained from the CFG), the dependence relations between the various tasks are defined. The detailed structure of each node in the task graph is as shown in the figure 5.2.

```
struct t_node {
        struct node *task;  /* first statement of the task */
        int task_no;
        int level;          /* level of the task in the graph */
        int processor;      /* processors assigned to the task */
        struct t_node *child;  /* pointer to the next task */
};
```

Figure 5.2: Structure of each node in the task dependence graph

Every node contains a pointer to the first statement of the corresponding task. The field `processor` indicates the number of processors assigned to a task by the scheduler. Since we do not consider inter task dependences, a linked list like structure adequately represents the relation between the various tasks. Every task contains a pointer to the next task to be executed. The structure can be easily modified to accomodate independent tasks.

We strictly prohibit jumps from outside a loop to inside or vice versa. Since we assume structured programming, we do not check for cycles formed by `gotos`. We hence discourage the user from using `gotos` frequently.

5.5 The Scheduler

A simple heuristic for defining tasks based on the language features is adopted. That is 'natural' boundaries in the program are used to define the tasks.

The scheduler at present, accepts the abstract syntax tree (IR) along with the control flow information supplied by Phase 1. A preliminary scan is done to identify the different tasks. In case of conditional statements (`if-then` and `if-then-else` constructs), all the branches are included in the same task. A loop nest is agin a task.

Since FORALL is the only type of parallel construct allowed in Fortran D, we safely assume that all tasks other than loops are strictly serial.

Processor assignment is then made to the tasks deterministically. OPTAL is employed to generate optimal processor assignments to loop nests. We use an array to store the intermediate results during the computation of the processor assignment. The number of *useful* processors with respect to a loop nest is automatically determined. This information is added to the nodes representing the loops in the IR.

OPTAL requires the size of all the loops to be known at compile time itself. We generate the three address representation [1] of the IR to get a rough estimate of the execution times of the loop bodies. We assume that the basic arithmetic and logic instructions provided by the processors take the same amount of time. An approach as naive as this may fail miserably if the loop bodies contain branches. In such cases (i.e loop bodies contain conditional branches), we consider the execution time of one of the branches. This again is not an accurate method for, only at run time can one know which branch the control will take. We hope that the control takes all its branches with equal probability and hence blindly choose one of the edges for estimating the execution times.

Since the task graph is a chain, there can be at most one task in every level. Hence we always assign processor 0 to execute serial tasks while each parallel task gets the first k processors, where k is the total number of processors assigned to the parallel task. [1]

5.6 The Back end

We then map the restructured program onto the underlying architecture. When data is not distributed, OPTAL decides the mapping. In case of data distributed programs, the data distribution patterns govern the process of mapping programs to the underlying hardware (see chapter 4). An interested user can identify from the SPMD code, the efficient data distribution patterns with respect to a loop nest.

Synchronizations are introduced at the end of every parallel loop and at the beginning of every parallel loop nest. In addition, prefetching of data is performed to improve the program performance. We assume that there is sufficient memory in each processor to hold the prefetched data, and hence pull out the array references outside the entire loop nests wherever possible. Two passes are made to make sure that data to be prefetched is neither modified nor left unused.

We choose to generate the SPMD code in a high level language. This is just a reverse process of the parsing stage. Along with the SPMD code we also give the restructured program back to the user with some extra information about the processor assignment. No sophisticated tools are provided for interactive compilation currently.

The various stages and their interfaces are as shown in the figure 5.3.

[1] processor field of each task would indicate this.

Chapter 6

Conclusions

In this chapter, we summerize the features of FRAMES as related to the process of partitioning and scheduling. We have already seen the advantages of having an automatic restructuring compilers which maps a parallel program to the underlying architecture efficiently. We now present some sample results obtained when FRAMES was tested on a number of benchmarking programs. We conclude with some suggestions for future work.

6.1 Results

FRAMES has been tested on some standard benchmarking programs. We have also tested FRAMES with some programs taken from the user space of *Convex C-220* machine at IIT Kanpur. We include the output generated by FRAMES for some programs.

6.1.1 Gauss Elimination

We now consider a traditionally coded gauss elimination program. All the loops in the nest can be parallelized. We present here, the parallelized version of the same in Fortran D. All the loops in the nest are normalized. Note that

```
dimension a(1000,1000)

c$dir max_trips 1000
forall i = 1,n
       a(i,i) = 1/a(i,i)
       c$dir max_trips 100
       forall j = 1,n-i
              a(j+i,i) = a(j+i,i)*a(i,i)
              c$dir max_trips 100
              forall k = 1,n-i
                     a(j+i,k+i) = a(j+i,k+i) - a(j+i,i)*a(i,k+i)
              endfor
       endfor
endfor
```

(a)

```
dimension a(1000,1000)

if ($_Pid .lt. 8) then
   SYNCHRONIZE
   do i=(($_Pid .mod. 8) .div. 8) * (n/1) + 1,
      MIN(((($_Pid .mod. 8) .div. 8 + 1) * (n/1)),n),1

      a(i,i)=1/a(i,i)
      do j=(($_Pid .mod. 8) .div. 2) * (n-i/4) + 1,
         MIN(((($_Pid .mod. 8) .div. 2 + 1) * (n-i/4)),n-i),1

         a(j+i,i)=a(j+i,i)*a(i,i)
         do k=(($_Pid .mod. 2) .div. 1) * (n-i/2) + 1,
            MIN(((($_Pid .mod. 2) .div. 1 + 1) * (n-i/2)),n-i),1

            a(j+i,k+i)=a(j+i,k+i)-a(j+i,i)*a(i,k+i)
         endo
         SYNCHRONIZE
      endo
      SYNCHRONIZE
   endo
   SYNCHRONIZE
endif
```

(b)

Figure 6.1: Gauss elimination program and the SPMD version

compiler directives have been used to assist the compiler in making scheduling decisions.

The SPMD equivalent of the above loop nest for an 8-processor multiprocessor is as shown in figure 6.1(b). Note that data dependences have suppressed prefetching. Because of the trapezoidal nature of the iteration spaces, the schedule obtained by FRAMES may not be necessarily *optimal*. (Note that FRAMES has assigned 2 processors to the innermost loop and 4 to the middle loop). Performance might have improved if all the 8 processors were assigned to the outermost loops. However, the compiler is not adept at doing this. The user could direct the compiler to do this by cleverly setting the `max_trips` directive. A dynamic scheme may possibly perform better.

If the compiler is unable to know even a rough estimate of the iteration space, then it assumes an upper bound of infinity (a very large number) and goes ahead with scheduling. In this example, FRAMES assigned all the 8 processors to the middle loop when no directives were given. This was because FRAMES assumed all the loop bounds to be infinite (and hence equal). In fact, with equal loop bounds, this is the optimal assignment.

6.1.2 Matrix multiplication

Matrix multiplication is another program which can be readily parallelized. Moreover, the restructured version of matrix multiplication contains the most general form of loop nests(i.e. hybrid) and hence was well suited for testing FRAMES.

The matrix multiplication program in Fortran D is as shown in figure 6.2. This program was obtained from the restructuring phase of FRAMES [3]. The program expresses all parallelism that can be extracted and no further parallelization is possible.

```
integer a(100,100),b(100,100),c(100,100)
forall i = 1,100
       forall j = 1,100
              c(i,j) = 0
       endfor
endfor
c the main loop begins
do k = 1,100
   forall i = 1,100
          forall j = 1,100
                 c(i,j) = c(i,j) + a(i,k) * b(k,j)
          enddfor
   enddfor
endo
```

Figure 6.2: Matrix multiplication program in Fortran D

Figure 6.3 shows the SPMD version of the matrix multiplication program. Note that only the loop bounds of the FORALLs have changed and all FORALLs have changed to DOs. Prefetching is possible here and hence is performed just before the loop nest. One can easily verify that assigning 2 processors to the loop i and 4 processors to the loop j of the main loop nest would yield the minimum execution time. The optimal assignment is not unique in this case. This is because the loops are perfectly nested. Assigning 2 processors to the innermost loop and 4 to the middle loop also yields an optimal assignment.

The conditional check at the beginning of every loop nest may seem to appear redundant. When the number of *useful* processors is less than the total number of available processors, this check prevents the processors unassigned from executing the loops. Note that every conditional check for the processor identifier (*$_Pid*) marks the beginning of a new task.

An interested reader can even make out the efficient data distribution patterns with respect to a loop nest, in case he wants to decompose the arrays. By

```
integer a(100,100),b(100,100),c(100,100)
if ($_Pid .lt. 8) then
   SYNCHRONIZE
   do i=(($_Pid .mod. 8) .div. 4) * 50 + 1,
      MIN((($_Pid .mod. 8) .div. 4 + 1) * 50,100),1

      do j=(($_Pid .mod. 4) .div. 1) * 25 + 1,
         MIN((($_Pid .mod. 4) .div. 1 + 1) * 25,100),1

         c(i,j)=0
      endo
      SYNCHRONIZE
   endo
   SYNCHRONIZE
endif
   c the main loop begins
if ($_Pid .lt. 8) then
   SYNCHRONIZE
   PREFETCH a
   PREFETCH b
   do k=1,100,1
      do i=(($_Pid .mod. 8) .div. 4) * 50 + 1,
         MIN((($_Pid .mod. 8) .div. 4 + 1) * 50,100),1

         do j=(($_Pid .mod. 4) .div. 1) * 25 + 1,
            MIN((($_Pid .mod. 4) .div. 1 + 1) * 25,100),1

            c(i,j)=c(i,j)+a(i,k)*b(k,j)
         endo
         SYNCHRONIZE
      endo
      SYNCHRONIZE
   endo
endif
```

Figure 6.3: SPMD version of Matrix multiplication program without data distributions

substituting the corresponding upper and lower bounds for the loop indices in the array index expressions that appear inside a loop nest, one can obtain the data distribution patterns. In the current example, one can see that the efficient data distribution patterns are `A(50,100)`, `B(100,25)` and `C(50,25)` with respect to the main loop. Figure 6.4 shows the SPMD equivalent of the matrix multiplication with the above data distribution.

In this case, one can see that in effect, each processor executes 50 iterations of loop i and 25 iterations of loop j. (Note that OPTAL also obtained the same result.)

For every optimal processor assignment, we can get the corresponding data distribution pattern. If 4 processors are assigned to the innermost loop and 2 processors to the middle loop, then the corresponding data distribution pattern would be `A(25,100)`, `B(100,50)` and `C(25,50)`.

6.1.3 The `dmxpy` routine (LINPACK)

The `dmxpy` routine again can be readily parallelized. We now consider this routine suitably modified to accomodate data distribution specifications as allowed in FRAMES (see figure 6.5).

The output of FRAMES for the above program is shown in the figure 6.6. Note that the loop bounds of the `FORALL` loop remain unchanged; but the loop body will now be executed conditionally. If we assume that all elements of the array `y` are written into, then each processor would execute a maximum of 13 iterations. In effect, the parallel loop is divided among the processors depending on the amount of data they possess. Prefetching is performed as usual.

Figure 6.7 gives the output of FRAMES for the `dmxpy` routine without data distributions, when compiled for an 8-processor multiprocessor. Note that if n1 is 100, then each processor will execute 13 iterations of the inner loop. This was

```
dimension a(100,100),b(100,100),c(100,100)
distribute a(2,1),b(1,4),c(2,4)

SYNCHRONIZE
do i=1,100,1
   do j=1,100,1
c
c check for locality
c
      if (i .le. ((($Pid .mod. 8) .div. 4) +1) * 50) .and.
         ((i. ge. (($Pid .mod. 8) .div. 4) * 50 + 1) .and.
         ((j. le. ((($Pid .mod. 8) .div. 2) + 1) * 25) .and.
         ((j. ge. (($Pid .mod. 8) .div. 2) * 25 + 1)

      c(i,j)=0
   endo
endo
SYNCHRONIZE
c
c the main loop begins
c
SYNCHRONIZE
PREFETCH a
PREFETCH b
do k=1,100,1
      do i=1,100,1
      do j=1,100,1
c
c check for locality
c
         if (i .le. ((($Pid .mod. 8) .div. 4) +1) * 50) .and.
         ((i. ge. (($Pid .mod. 8) .div. 4) * 50 + 1) .and.
         ((j. le. ((($Pid .mod. 8) .div. 2) + 1) * 25) .and.
         ((j. ge. (($Pid .mod. 8) .div. 2) * 25 + 1)

            c(i,j)=c(i,j)+a(i,k)*b(k,j)
         endo
      endo
endo
SYNCHRONIZE
```

Figure 6.4: SPMD version of Matrix multiplication program with data distributions

```
dimension x(100),y(100),m(100,100)
distribute x(8),y(8),m(4,2)
c
c main loop begins here
c
jmin = j+16
do j = jmin, n2,16
   forall i = 1,n1
          y(i) = y(i) + x(j-15)*m(i,j-15) + x(j-14)*m(i,j-14) + x(j-13)*m(j-13)
                 + x(j-12)*m(j-12) + x(j-11)*m(j-12) + x(j-11)*m(i,j-11)
                 + x(j-10)*m(j-10) + x(j-9)*m(j-9) + x(j-8)*m(i,j-8)
                 + x(j-7)*m(j-7) + x(j-6)*m(j-6) + x(j-5)*m(i,j-5)
                 + x(j-4)*m(j-4) + x(j-3)*m(j-3) + x(j-2)*m(i,j-2) + x(j-1)*m(i,j-1)
   endfor
endo
```

Figure 6.5: The dmxpy routine with data distributions

exactly the situation when data was distributed.

6.2 Limitations of FRAMES

FRAMES leaves a lot of scope for further improvement. We list below some of the limitations of FRAMES and suggest some improvements that could be easily done.

- Since a heuristic approach is taken for partitioning a program, we may lose out a large fraction of parallelism that could be present in constructs other than loops. A better performance could, however, be realized by providing a suitable set of compiler directives to define tasks.

- A serious drawback of FRAMES is its policy to perform static processor assignments to the tasks. Had we delayed the scheduling decisions till run time, we could have possibly exploited a greater degree of parallelism. By including dynamic scheduling strategies into FRAMES, one could compare

```
dimension x(100),y(100),m(100,100)
distribute x(8),y(8),m(4,2)
c
c main loop begins here
c
if ($_Pid .lt.  1) then
jmin = j+16
endif
SYNCHRONIZE
PREFETCH x
PREFETCH m
do j = jmin, n2,16
   do i = 1,n1
c  check for locality
      if (i .le.  ((($_Pid .mod.  8) .div.  1) +1) *13)
         .and.  (i .ge.  ((($_Pid .mod.  8) .div.  1) *13 +1)

         y(i) = y(i) + x(j-15)*m(i,j-15) + x(j-14)*m(i,j-14) + x(j-13)*m(j-13)
                + x(j-12)*m(j-12) + x(j-11)*m(j-12) + x(j-11)*m(i,j-11)
                + x(j-10)*m(j-10) + x(j-9)*m(j-9) + x(j-8)*m(i,j-8)
                + x(j-7)*m(j-7) + x(j-6)*m(j-6) + x(j-5)*m(i,j-5)
                + x(j-4)*m(j-4) + x(j-3)*m(j-3) + x(j-2)*m(i,j-2) + x(j-1)*m(i,j-1)
   endo
endo
SYNCHRONIZE
```

Figure 6.6: The SPMD version of the dmxpy routine with data distributions

the performance of the two.

- At present FRAMES can only exploit parallelism within a task. FRAMES may be able to perform better if inter task parallelism is also considered. Extending FRAMES to handle independent tasks could be an interesting work.

- A sophisticated approach to estimate task execution times is needed to make more accurate scheduling decisions. Execution times of tasks that contain branches may be estimated by considering those branches which are visited more frequently. This information could be obtained from

```
dimension x(100),y(100),m(100,100)
c
c main loop begins here
c
if ($_Pid .lt.  1) then
jmin = j+16
endif
if ($_Pid .lt.  8) then
SYNCHRONIZE
PREFETCH x
PREFETCH m
do j = jmin, n2,16
   do i = (($_Pid .mod.  8) .div.  1) *(n1 .div.  8) +1,
      MIN((((($_Pid .mod.  8) .div.  1 + 1) * (n1 .div.  8)),n1),1
      y(i) = y(i) + x(j-15)*m(i,j-15) + x(j-14)*m(i,j-14) + x(j-13)*m(j-13)
             + x(j-12)*m(j-12) + x(j-11)*m(j-12) + x(j-11)*m(i,j-11)
             + x(j-10)*m(j-10) + x(j-9)*m(j-9) + x(j-8)*m(i,j-8)
             + x(j-7)*m(j-7) + x(j-6)*m(j-6) + x(j-5)*m(i,j-5)
             + x(j-4)*m(j-4) + x(j-3)*m(j-3) + x(j-2)*m(i,j-2) + x(j-1)*m(i,j-1)
   endo
   SYNCHRONIZE
endo
SYNCHRONIZE
endif
```

Figure 6.7: The SPMD version of the dmxpy routine without data distributions

previous test runs.

- Regarding data distributions, FRAMES can handle only those arrays that are distributed blockwise. An extension to other distribution patterns can be easily made. Finding an efficient processor assignment to a program with respect to a data decomposition is still an area of research.

Appendix A

Features of Fortran D

Fortran D (D stands for data,decomposition,distribution) is a parallel version of Fortran 77 with a rich set of data decomposition specifications. Fortran D is well suited for writing data parallel programs. The language has been so designed that a sophisticated compiler can produce efficient programs for different parallel architectures. In particular, Fortran D is best suited for programming distributed memory machines, where data decomposition forms an integral part of data parallel programs. This chapter presents some details of Fortran D, especially its strategy for expressing data parallelism and mapping it to the underlying architecture. An inquisitive reader is referred to [6], [8].

A.1 Expressing Data parallelism

Fortran D is designed to support two fundamental stages of writing a data parallel program; problm mapping and machine mapping. Problem mapping deals with how arrays should be aligned with respect to one another. This is largely independent of any machine considerations. Machine mapping translates the problem onto the finite resources of the machine. i.e. machine mapping specifies how arrays should be distributed onto the actual machine. This is

dependent on the machine details like general topology, number of processors, communication mechanisms, size of local memory etc.

In Fortran D, the user has to specify data parallelism in terms of these two levels. The `ALIGN` statement describes a problem mapping while the `DISTRIBUTE` statement defines the machine mapping.

A.2 Problem mapping

In Fortran D, every problem mapping is associated with a name which needs to be declared using the `DECOMPOSITION` statement. `ALIGN` maps the arrays in the program to the decompositions. An array can be mapped to many decompositions within a program, but at a time an array is mapped to only one decomposition.

`DECOMPOSITION` declares the name, dimensionality and size of a decomposition for later use. A decomposition is an abstract problem and no storage is allocated to it.

```
DECOMPOSITION A(N,N)
```

Here `A` is declared to be a two dimensional (`N` $\times$ `N`) decomposition with the elements in each dimension indexed from 1 to `N`.

The `ALIGN` statement maps the arrays to decompositions. More than one array can be mapped on to a decomposition. Arrays mapped to a single decomposition are aligned with each other.

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X(I,J) with A(I,J)
```

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X with A
```

```
REAL X(N)
DECOMPOSITION A(N)
ALIGN X(I) with A(I+1)
```

(a). An offset of 1

```
REAL X(N)
DECOMPOSITION A(N)
ALIGN X(I) with A(2*I)
```

(b). A stride of 2

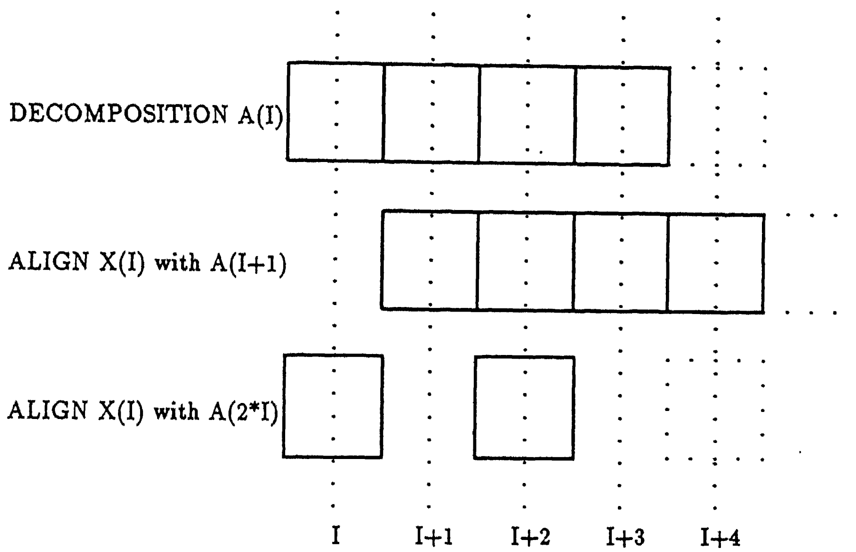

Figure A.1: Alignment of arrays with decompositions

In the above example, X is mapped exactly onto the equivalent dimensions in the decomposition A. We could have had alignment offsets or strides as shown in figure A.1.

Strides can be used in combination with offsets. Fortran D also allows permuting the dimensional alignment between arrays and decompositions.

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X(I,J) with A(J,I)
```

In this example, the transpose of X is mapped to the decomposition A, as indicated by the reversed place holders I and J. It is not necessary that there

should be a dimensional match between the arrays and decompositions. In the following example, the first dimension of X is mapped onto the decomposition A. In other words, each row of X is mapped to an individual element in the decomposition A.

```
REAL X(N,N)
DECOMPOSITION A(N)
ALIGN X(I,J) with A(I)
```

In Fortran D, just a part of the array can also be mapped to a decomposition.

```
REAL X(2*N)
DECOMPOSITION A(N)
ALIGN X(I) with A(I) range (1:N)
```

In the above example, only the first N elements of X are mapped onto A.

A.3 Machine mapping

DISTRIBUTE statement specifies the mapping of a decomposition to the physical machine. The distribution selected affects the ability of the compiler to minimize communications and load imbalance for the resulting program. Physical machine characteristics such as the number of processors, size of the local memory and communication costs should be taken into consideration while defining efficient distributions. In Fortran D, a DISTRIBUTE statement specifies machine mapping for exactly one decomposition. All arrays mapped to the decomposition are distributed.

The DISTRIBUTE statement takes the name of a decomposition and assigns an *attribute* to every dimension of the decomposition. The attribute describes the mapping of data in that dimension. Only one attribute can be assigned to a dimension of a decomposition. Attributes in each dimension are independent and may specify regular or irregular distributions. The symbol '*' is used when a particular dimension is not distributed. Dimensions not distributed are assigned to all processors.

```
REAL X(N)
DECOMPOSITION A(N)
ALIGN X(I) with A(I)
DISTRIBUTE A(attribute)
```

Distributions thus describe the assignment of data to the underlying processor array.

Regular distributions allowed in Fortran D are as follows.

1. BLOCK
2. CYCLIC
3. BLOCK_CYCLIC

If there are N elements in a decomposition and p processors, then

1. BLOCK divides the decomposition into contiguous chunks of size $\lceil N/p \rceil$, assigning one block to each processor.
2. CYCLIC defines a round-robin division of the decomposition, assigning every p-th element to the same processor.

3. BLOCK_CYCLIC is somewhat similar to CYCLIC. This attribute takes a parameter M. Here contiguous chunks of size M are distributed cyclically.

Fortran D allows the user to specify how many processors should be assigned to each dimension of a decomposition. The product of the number of processors in all dimensions should be less than or equal to the total number of available processors.

```
DISTRIBUTE A(BLOCK(4),BLOCK(2))
```

In the above example, the first dimension of A is distributed among 4 processors while the second dimension is distributed among 2 processors.

In Fortran D, irregular data distributions can be specified through the use of a mapping array. An example for implementing irregular data distributions is given below.

```
REAL X(N)
INTEGER MAP(N)
DECOMPOSITION REG(N),IRREG(N)
ALIGN MAP with REG
ALIGN X with IRREG
DISTRIBUTE REG(BLOCK)

... set values of MAP by some algorithm ...

DISTRIBUTE IRREG(MAP)
```

Here the elements of MAP should be set to integers between 1 and p, where p is the total number of processors.

Since nature in which arrays are used in different segments of the program varies, data mappings change accordingly. Dynamic realignment and redistribution is therefore needed to reduce data movement. Fortran D hence supports

dynamic data decompositions. Fortran D considers ALIGNs and DISTRIBUTEs as executable statements rather than declarations. By including them at different points in the program, dynamic data distributions can be achieved. The following figure illustrates an example where data is distributed dynamically.

```
REAL X(N)
DECOMPOSITION A(N)
ALIGN X(I) with A(I)
DISTRIBUTE A(BLOCK)
DO I = 1,N
   X(I) = ...
ENDDO
...
ALIGN X(I) with A(I+1)
DISTRIBUTE A(CYCLIC)
DO I = 1,N
   ...  = X(I+1)
ENDDO
```

The compiler has to perform sophisticated analysis to handle dynamic data distributions. Reaching deomposition analysis (somewhat similar to reaching definition analysis) can be performed to handle dynamic data distributions [8].

Bibliography

[1] Aho, A. V., Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.*, Addison-Wesley, 1986

[2] Allen, R. and Kennedy, K. *Automatic Translation of Fortran Programs to Vector Form*, ACM Transactions on Programming Languages and Systems, Vol. 9, No. 4, Oct. 1987, pp. 491-542.

[3] Ananda, R. *Fortran D Parallelizing Compiler: Restructuring Phase*, MTech thesis report, Dept. of CSE, IIT Kanpur, Feb. 1993.

[4] Banerjee, U. *Dependence analysis for Supercomputers*, Kluwer Academic, 1988.

[5] *Convex Fortran Optimization Guide*, Convex Computer Corporation, 1990.

[6] Fox, G., Hiranandani, S., Kennedy, K., Koebel, C., Kremer, U., Tseng, C. and Wu, M. Fortran D *Language Specification*, Tech. Rep. TR 90-141, Dept. of Computer Science, Rice University, Dec. 1990.

[7] Gornish, E. H., Granston, E. D. and Viedenbaum, A. V. *Compiler-directed Data Prefetching in Multiprocessors with Memory Heirarchies*, Proceedings of the 1990 International Conf. on Supercomputing, ACM SIGARCH Vol. 18, No. 3, Sep. 1990, pp. 354-368.

[8] Hiranandani, S., Kennedy, K. and Tseng, C. *Compiling* Fortran D *for MIMD Distributed Memory Machines*, Comm. of the ACM, Vol. 35, No. 8, Aug. 1992, pp. 66-80.

[9] Hummel, S. F., Schonberg E. and Flynn L. E. *Factoring, A Method for Scheduling Parallel Loops*, Comm. of the ACM, Vol. 35, No. 8, Aug. 1992, pp. 90-101.

[10] Hwang, K., and Briggs, F. A. *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[11] Kasahara, H. and Seinosuke, N. *Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing*, IEEE Trans. on Computers, Vol. C-33, No. 11, Nov. 1984.

[12] Nori, K. V., Kumar, S. and Kumar, M. P. *Retrospection of the PQCC Compiler Structure*, Proceedings of Foundations of Software Technology and Theoritical Computer Science, Dec. 1987, LNCS Vol. 287, Springer-Verlag, pp. 500-527.

[13] Singh, R. and Purohit, V. *A Tool for Vectorizing Sequential Programs*, BTech project report, Dept. of CSE, IIT Kanpur, 1991.

[14] Singh, R. MTech thesis report (to be published), Dept. of CSE, IIT Kanpur,1993.

[15] Polychronopoulos, C. D. *Parallel Compilers*, Kluwer Academic Publishers, 1987.

[16] Stone, H. S., *Multiprocessor Scheduling with the aid of Network Flow Algorithms*, IEEE trans. on Software Engg., Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

[17] Wang, C. and Wang, S., *Efficient Processor Assignment Algorithms for Executing Nested Parallel Loops on Multiprocessors*, IEEE Trans. on Parallel and Distributed Systems, Vol. 3, No. 1, Jan. 1992, pp. 71-82.

[18] Wolfe, M. J., *Optimizing Supercompilers for Supercomputers*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1982.